SYSTEM 250

Papers Presented at I.C.C. Conference Washington D.C. Oct. 1972

PLESSEY TELECOMMUNICATIONS RESEARCH LIMITED,
TAPLOW COURT,
TAPLOW, MAIDENHEAD,
BERKSHIRE,
ENGLAND.

CONTENTS

A CAPABILITY ORIENTED MULTI-PROCESSOR SYSTEM FOR REAL-TIME APPLICATIONS

D. C. Cosserat

Plessey Co. Ltd.
Liverpool, England

Summary

The system under consideration is a multi-processor, multi-storage module configuration adapted to the processing and fault security requirements of such real-time applications as telephone switching, message switching, and radar systems control. Each processor accesses store independently and asynchronously and each region of store to which it has immediate access is bounded by an addressing structure known as a Capability. The capability has a dual purpose. It acts as a protection mechanism against hardware and software failure; and it defines a logical unit of contiguous storage space (a "segment") out of which all operating system and application data structures are built. A segment may contain either data or capabilities permitting a list structure of interconnected segments to be established.

Each processor executes instructions contained in data segments, achieving linkage from segment to segment, and in so doing manipulates the data structure appropriately. One particularly significant feature of the system is that it is possible for a running program to make copies of capabilities which it can store arbitrarily into the data structure. The operating system reduces essentially to a series of 'protected subroutines', each subroutine possessing just the capabilities required to gain appropriate access to the data structure. There exists therefore a graded approach to storage protection and a complete lack of the visual division into 'special' and 'normal' modes of machine hardware operation.

Introduction

Computer systems are characterised by their ability to provide 'general purpose' solutions to specific logical problems. In the telecommunications field, in particular, the computer may be used as a centralised control mechanism which replaces the logical functions often formerly provided by distributed hardware devices. Thus, for example, the centralised computer system can be used to control telephone switching hardware which itself contains little or no sequential logic either from a mechanical or electronic point of view. Similarly a computer may be used to provide automatic routing of messages in a message switching network; automatic information retrieval, computation, and display in an air traffic control environment; centralised control of industrial processes; network control of distributed systems such as electricity and gas grids; area control of road traffic schemes; etc.

Three important factors relevant to these systems are security, growth and obsolescence. Real-time systems whose operation affects a large number of human beings must be capable of withstanding long periods between system failures. In some cases this requirement arises from an economic or strategic need (in telephone switching systems, for example) and in others (such as air traffic control) human lives are directly involved. The second factor arises because telecommunications networks have traditionally been designed so that increases in size and 'traffic' carrying capacity can be accommodated over a period of years. Thirdly the nature of telecommunications networks and, in particular, the amount of capital investment required implies that systems installed today should not become rapidly obsolescent.

Traditionally, the kind of computers that have been applied to these real-time control tasks have emerged from two quite different stables; on the one hand, system designers have made attempts to adapt computing equipment developed in the data processing environment to the requirements of real time control, and on the other hand engineers who have experience of existing electromechanical and electronic techniques have tried their hand at producing computer systems. This dichotomy of discipline has led in the past to a polarisation of ideas on how real-time centralised control systems should be built. As a broad generalisation it might be said that the computer engineers have failed to design systems which have the security and expandability features so characteristic of telecommunications systems, whilst the telecommunications engineers have failed to design systems which promote to the full the control flexibility afforded by software technology.

In order to illustrate the problems confronting the computer system designer in this field it is useful to select a particular case for analysis. The case chosen here is that of the telephone switching control problem because it represents a particularly comprehensive example of conflicting requirements. Designers who are interested in other real time application areas will, however, recognise many analogies with their own problems.

In the case of telephone switching control, it was thought for a long period of time that the major problem involved was the tricky technological one of designing a centralised control device to obey the necessary logical steps to control the switches involved in setting up a circuit from one telephone subscriber to another. This was obviously the immediate and central task and it was tackled in a variety of different ways. Some solutions involved hardware-wired logic as the means of centralised control, others involved the use of a computer-like device which fetched instructions from a read-only store, and yet others utilised a true computer configuration in which a processing unit fetched instructions from and modified data in, a read/write store. At the beginning, it did not really seem to matter very much which particular system was chosen because the central problem of switch control was identical in all cases, and often the decision as to which system to adopt depended on the design experience and background of the individuals concerned.

On the basis of a computer's ability to set switches in a telephone network, it was also arguable that there were no obvious advantages in any of these approaches over the previous electromechanical systems. Certain peripheral advantages were said to accrue such as 'system flexibility', but what exactly did this mean, and how was such a nebulous term to be quantified? Nowadays, it is possible to enumerate a number of facilities provided by a computer controlled telephone switching system. For example,

automatic fault diagnosis
centralised maintenance
network monitoring
automatic accounting
integrated manual assistance facilities
special subscriber facilities

All these factors were, of course, recognised by the advocates of centralised control, but they were regarded as a bonus to the more immediate problem of the switching task itself. A closer examination of these and other similar facilities leads, however to a new concept: a centralised computer system for telephone switching control must be regarded as an administrative unit which interfaces primarily with the staff of the telephone administration. Except for the provision of special facilities, the interface to the telephone subscribers is of secondary significance since it necessarily remains very much the same as in all previous systems.

In a computer controlled telephone network all the above facilities can be provided in a centralised manner. Fault diagnosis and maintenance can be handled by a relatively small staff via interactive video-displays; network monitoring programs can be similarly controlled by a few staff at a centralised location. Automatic accounting software can remove the human data preparation link, passing metering information from individual calls into a form suitable for the direct printing of accounts. Manual board operators can communicate via similar video displays on which all information pertinent to the call is recorded. The operator has sufficient control to achieve the required objective without the necessity of any administrative overheads, such as the filling out of dockets: instead, the system records the call details and cost automatically and routes it directly through to the accounting software system. This approach implies a unification of system design and, where necessary, the derivation of simple and standard ergonomic interfaces with those staff who control it. The activities mentioned above are summarised in Figure 1.

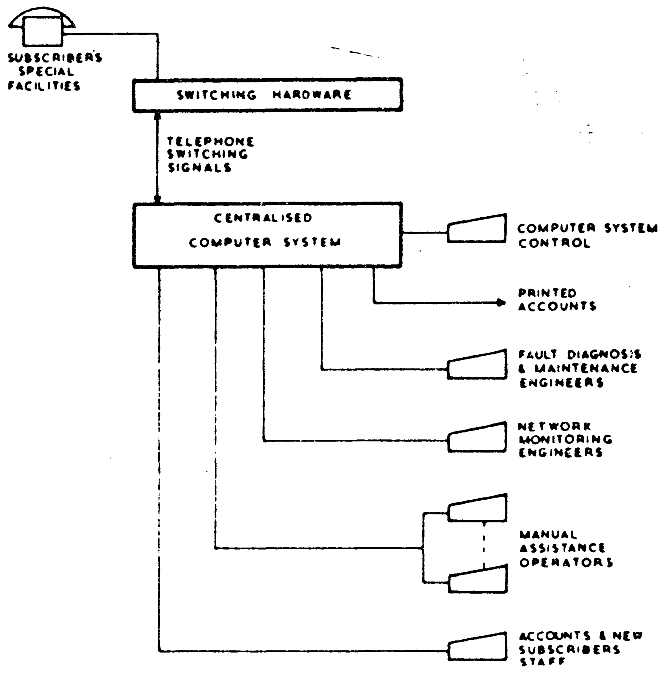
SUBSCRIBER'S SPECIAL FACILITIES
SWITCHING HARDWARE
TELEPHONE SWITCHING SIGNALS
CENTRALISED COMPUTER SYSTEM
COMPUTER SYSTEM CONTROL
PRINTED ACCOUNTS
FAULT DIAGNOSIS & MAINTENANCE ENGINEERS
NETWORK MONITORING ENGINEERS
MANUAL ASSISTANCE OPERATORS
ACCOUNTS & NEW SUBSCRIBERS STAFF

Fig. 1 ORGANISATION OF A CENTRAL PROCESSING SWITCHING UTILITY

Once the centralised control system is regarded as an aid to administration of the network, and once it is understood that it is here that the real economic advantages lie, the requirements of the centralised computing system necessary to support such activities become very different from those needed to handle the switching problem itself. In particular it is clear that wired logic or read-only program storage systems will not provide the necessary flexibility; and it is clear also that comprehensive software facilities are required to sensibly tackle the application requirements of what has become a real-time system with multi-access control.

The General Purpose Computer

It is clear from the foregoing telephone switching example that the problems of large and comprehensive central control systems are not amenable to solution either by dedicated telecommunications processors or by existing computers designed for the data processing environment.

Here is a list of some of the more obvious and important requirements:

- ABILITY TO RUN REAL-TIME PROGRAMS
- MULTI-PROGRAMMING FACILITIES
- MULTI-ACCESS FACILITIES FOR MAN/MACHINE CONTROL
- STRICT INFORMATION PROTECTION BETWEEN PROCESSES
- THE CONVERSE ABILITY TO SHARE INFORMATION BETWEEN PROCESSES WHERE REQUIRED
- HARDWARE EXPANDABILITY IN INDEPENDENT INCREMENTS OF STORAGE AND PROCESSING POWER
- AUTOMATIC RECONFIGURATION FOLLOWING SYSTEM FAILURE
- FLEXIBLE INTERFACING TO DISTRIBUTED EQUIPMENT AND TO MAN/MACHINE DEVICES

In the light of these requirements, and the fact that existing data processing systems do not match up to all of them, we prefer to reserve the term 'general purpose computer' for a system which meets all these characteristics. Given this definition, it becomes clear that neither existing data processing systems nor telecommunications processors can in any sense be regarded as 'general purpose'.

Design Considerations

In order to achieve the above design objectives, a combination of hardware and software technological innovations must be employed. One particularly important feature involves system expansion.

The computer configuration must be capable of expanding in two important aspects: there should be no practicable limits on the size of the fast store; and there should be as wide a range as possible of processing power. In each of these cases, the hardware should be expandable in reasonably small increments so as to permit a smooth rate of increase in capital investment in the system. It is particularly important that increases in storage and processing power can be achieved independently, since there is no obvious correlation between one and the other over a wide range of possible systems. Therefore a true multi-processor system which can contain a variable number of processing units and a variable number of storage modules is the ideal for the application.

The system must be resilient against both solid and transient hardware failures, and similarly against software bugs (which have many of the characteristics of transient hardware failures). This requirement means in practice that the system should be capable of automatic reconfiguration (i.e. switching out the failed hardware module) and recovery (i.e. the ability to return to the execution of a coherent program and data base).

The general purpose computer must also be capable of interfacing freely with a wide range of distributed telecommunications equipment, which may be remote from or local to the computer itself, and also must interface with man/machine devices such as video-displays and other computer peripheral devices.

Design Conflicts

The above remarks are addressed to some of the more obvious and important features of the general purpose computer. But some of the design requirements conflict and it is necessary to examine these conflicts in some detail.

The first design conflict arises from the requirement on the one hand to use the processors in a 'work sharing' mode to meet the requirements of a multi-programming, multi-processor system and on the other to respond quickly to interrupts generated by signals from the real-time system under control. Each processor must inherently be capable of obeying any program steps in the system (a functional approach involving the division of processors to specific tasks would conflict with the multi-processing requirement and with the need to expand the system with little software re-organisation).

The execution by a processor of a program is conventionally termed a process. In a multi-processor system there can clearly be as many processes in simultaneous execution as there are processors, but there may be an undefined number of additional processes which are blocked awaiting logical events or are freed but have no processor on which to run. When a process runs, the processor contains in its hardware registers information relevant to that process and when the process blocks, that information must be stored away. In a processor with several registers, the storing of their contents may involve many store accesses. An interrupt is caused by an event in the outside world which raises a signal into the computer system. This causes the processor to cease its present activity (i.e. to temporarily block the running process) and to execute an 'interrupt process' instead. The changeover from one process to the other involves the storing and loading of registers and hence there is a processing time 'overhead' on each interrupt.

In a single processor environment, this problem is often solved by the use of a second set of registers reserved for the interrupt process. In a 'work-sharing' multi-processor system this approach is not possible because the interrupted process is still logically free to run and may be picked up immediately by another processor. In this situation, the information concerning the process which is stored away in the first processor's register set is completely inaccessible to the second processor. Since the common medium of communication between processors is the store, it follows that the register information of an interrupted process must be written to store where it may be retrieved subsequently by another (or the same) processor. This register storing overhead is a theoretical limitation on a true multi-processor system and as such represents a design conflict between the attributes of such a system and the requirement to respond quickly to interrupts.

The second design conflict arises because of a potential fault security hazard in the universal sharing of store by all the processors in a multiprocessor system. In order to provide a system which is expandable in independent increments of storage and processing power it is necessary to abandon the usual concept of a computer as 'processing unit plus store'. The corollary of this divorce between processor and dedicated store is that a storage module belongs to no processor in particular and is equally accessible from all processors. Such a system organisation achieves the hardware modularity constraint at the expense of another; namely, resilience against store corruption which could lead to undetectable system failure. The concept of a multi-store, multi-processor system, which is so attractive from the point of view of modularity, is wide open to the possibility of storage corruption from any processor that fails transiently or permanently. Therefore there is a design conflict between 'equal availability of all storage locations' and 'potential damage to storage contents by a faulty processor'.

System 250 - A General Purpose Computer

An attempt has been made to embody the general design principles described above in the Plessey SYSTEM 250 central processing system. This system is designed for precisely the range of applications described and in particular for the control of administration and switching functions in a telephone switching environment. The design includes the following features which are compatible with and a consequence of the requirements of a general purpose computer.

Firstly, the hardware is designed as a multiprocessor, multi-storage module configuration as represented in Figure 2. Each processor may access any storage location in any store module over its own bus. Thus the modularity and incremental expansion requirements of the general purpose computer are satisfied.

Secondly, each processor is capable of detecting a range of hardware fault indications which will cause a fault interrupt to be automatically generated. The processor discontinues execution of the current process and switches to a fault interrupt process instead. The instructions obeyed by this process are, of course, fetched from store in the usual way but, should a subsequent fault interrupt be generated in that processor during the time that it is executing the fault interrupt process, the processor steps to the next storage module and recommences the process by fetching instructions from it. Thus, a failure in the storage module itself or corruption of its contents does not cause a permanent failure of the processor which received the original fault interrupt. The essential hardware mechanisms are therefore provided as a basis for an automatic reconfiguration software system.

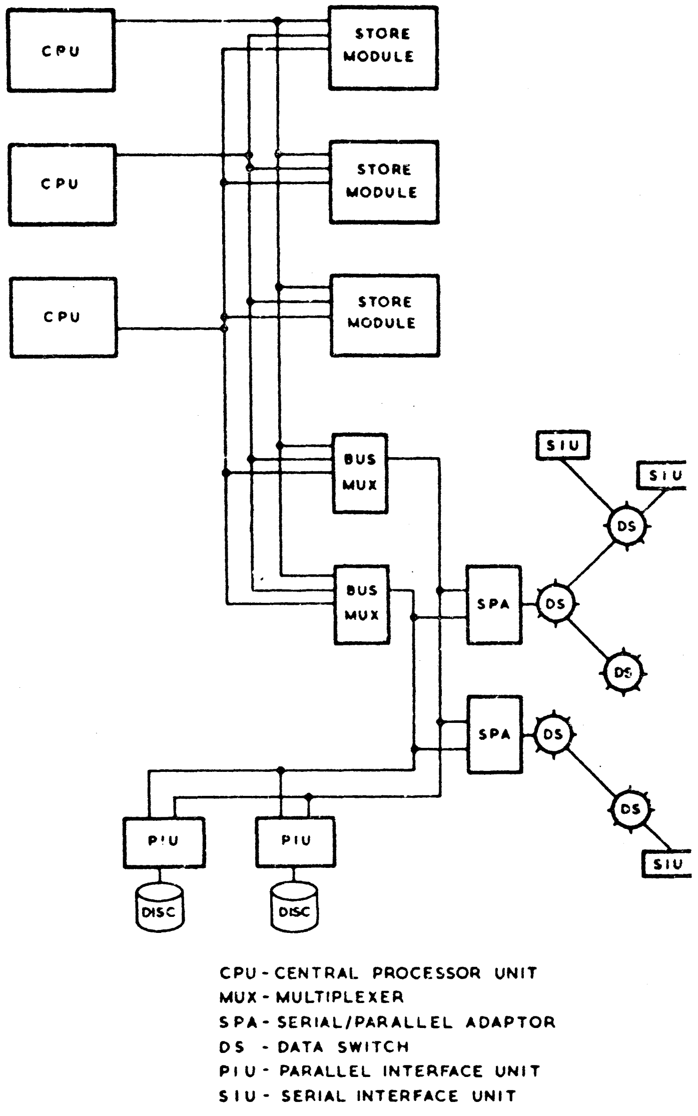

FIG. 2 TYPICAL HARDWARE CONFIGURATION

Thirdly, a flexible interfacing medium enables the system to be adapted to a wide range of peripheral equipment. The interconnection network is in the form of a bit-serial medium which transfers 'messages' between computer system and the periphery in both directions. A standard serial interface makes it possible to build a structure of 'primary' and 'secondary' electronic switches to suit a particular configuration and to interface simply to man/machine interactive devices. This satisfies the requirement that a general purpose computer should be connectable in a flexible manner to a wide range of telecommunications and other equipment.

Given the above general features, it is now necessary to describe how the system overcomes the two basic design conflicts mentioned above.

Interrupts

The first conflict concerns the incompatibility between a true multi-processor system and the 'overheads' involved in servicing an interrupt.

An examination of this problem led inevitably to a study of the history of interrupt mechanisms. Very early computers which had no interrupt systems suffered from the major disadvantage that tests of peripheral device status had to be inserted into the program at regular intervals. A natural consequence of this was the design of interrupt hardware which performs this testing between the execution of each instruction. When an interrupt occurs, the processor ceases to execute its current process and switches automatically to the execution of an interrupt process. In the case of input data, this process typically does nothing more than place the information in a software organised queue in store. This queue is unloaded by a normally scheduled process and the information is then analysed and used as appropriate for the application. In its role of executing the interrupt process, the processor is behaving essentially as a hardware queueing mechanism and can therefore be replaced by a hardware queue in the interface between the serial medium and the processing system. In SYSTEM 250 the unit known as the Serial/Parallel adaptor (see Figure 2) performs this function and, typically, can queue up to sixteen messages to and sixteen messages from the serial medium. A normally scheduled process unloads the messages from the hardware queue directly.

Another input/output requirement involves the transfer of data from magnetic backing storage devices such as drums and discs. Since it is usually uneconomic in conventional systems to withstand an interrupt for each word or character transferred, the standard approach is to use channelling hardware which moves data directly between 'burst mode' devices and store. During the transfer the processor is free to perform its usual functions and only receives an interrupt from the channelling hardware when the data transfer is complete. The usual characteristic of channelling hardware is that it is both elaborate and expensive; and it is, perhaps, unfortunate that there appears to be a tendency amongst computer designers in the direction of more elaboration and more expense. Some channellers are completely hard-wired, others obey special instructions fetched from store and begin to look very much like special purpose computers. SYSTEM 250 has taken this trend to its logical conclusion and utilises the standard processor module as a channeller. This approach has two very important advantages. Firstly, in a secure system there is no additional 'sparing' problem. Whereas it would have been necessary to provide a second channeller as a fault security backup, the additional processor now required for channelling work can share the existing spare processor(s) required to maintain processing security. Thus, in the majority of system configurations, the cost comparison is between one processing module and two channeling modules. Secondly, there is now no requirement for interrupt generation at the end of a data transfer because the processor itself can continue to process what would have been the interrupt response routine.

The two features of SYSTEM 250 described above, namely hardware message queueing and the use of the processors as channellers, has abolished the need for external interrupts and has therefore resolved the conflict between interrupts and the efficient operation of a true multi-processor system. Additionally it has provided a cheap and conceptually elegant form of input/output control.

It should be observed, in conclusion, that there are still three mechanisms in a SYSTEM 250 processing module which can force a change of process: the first is due to a program trap condition; the second occurs when the processor's interval timer clock value reaches zero; and the third is due to the occurrence of an internally detected fault condition. Although these conditions may colloquially be referred to as 'interrupts', the common characteristic is that in no case is the condition externally imposed. The abolition of inter-processor and device-processor interrupt lines has a significant effect on the security of the hardware and makes it simpler to isolate processors and peripheral units following hardware failure.

It can be seen from the above discussion that the requirements of the general purpose computer are highly interactive. Both security requirements and the need for interrupt free operation of the multi-processor system affect the input/output economics in an unexpected way. By turning these conflicting constraints to advantage rather than by adopting some conventional compromise solution, it has proved possible to realise a simpler and more economically attractive solution to the problem.

Storage Protection - The Capability

The second design conflict which must be solved in the quest for the general purpose computer concerns the potential for storage corruption in a true multi-processor system. Two separate fears may be expressed on this subject. Firstly, there is the fear that processors which have access to the whole of the storage system may corrupt the program and read-only data held there. This will almost certainly result in a catastrophic failure of the system with instructions and data constants coverted to random values. This problem has led some designers to criticise the nature of an alterable store for critical real-time applications and to suggest that the older schemes of wired logic processors or of computers with their own dedicated storage modules are more adapted to the requirements.

As will be shown below, the nature of the problem is not so much the volatility of the storage medium itself as a lack of discipline on the part of the processors in their attempts to access it. It is this latter aspect to which attention has therefore been turned in an attempt to preserve the general purpose features of a freely alterable storage system.

The second fear is that, even in a system where read-only and read-write information is strictly segregated, there is still the possibility that faulty processors will obey random instruction sequences, attempt to obey read-only data as instructions, and alter read-write data values to which the currently obeyed program has no logical access. In short, even in a partitioned system of this type there is still much scope for corruption of store and therefore of system failure.

Solutions to this information protection problem typically involve the use of base-limit protection registers which partition the store into a number of contiguous regions or segments. Further protection measures may be applied to restrict access, such as the 'rings of protection' scheme suggested by Graham (reference 1). What is required, then, is a mechanism which permits the programmer precisely to define those data structures which will be made accessible to a running process and, by default, those which will not. There must be no system feature which prevents information sharing where this is logically required, and conversely, no system feature which permits information sharing where this is not logically necessary.

The solution which has been chosen in SYSTEM 250 involves the provision of hardware protection features which permit a given running process to access only those regions of store that the programmer originally intended. This is achieved by means of a universal segment identifier known as a capability. A capability is an invarient address which defines (a) the absolute location of a segment of storage, (b) the length of the segment, and (c) the kind of access permitted (read-only, execute only, read-write, etc.). What distinguishes a capability from a traditional base-limit protection address is that it can be freely copied by the running process itself (i.e. it can be loaded into a machine register and can be stored into a storage location), but that its contents can in no way be altered. The concept of a capability originated in the work of Dennis and Van Horn (reference 2), and was proposed in the present freely copiable form by Fabry (reference 3). The use of the capability mechanisms in SYSTEM 250 has already been described in detail elsewhere (reference 4) and no further elaboration will be attempted here.

The essential feature of a capability is its ability to permit the currently running process access to carefully controlled and logically necessary regions of the store. The hardware is arranged so that there is no way in which a process can manufacture data patterns and convert them into capabilities; therefore, there is no way in which it can gain access to, and possibly corrupt, other regions of the store. This, then, is what is required in order to prevent the collapse of a multi-processor system due to storage corruption by a single processor.

The corollary of the above is that, when faults do occur in a processor, the strict control of base, limit and access conditions assist the system greatly in the fast detection of failures.

Software Implications - The Operating System

The capability was primarily developed as a mechanism for storage segmentation and information sharing rather than for hardware protection. Of course, its protection features were always recognised in the context of protection between programs and it is here that the major software implications lie.

One of the criteria of the general purpose computer is that it should be capable of information sharing. This is a critical requirement for many real-time applications where many transactions are represented by processes sharing a common data base, but may also be considered a general requirement of any computer system in which multi-access facilities are required. Computer systems which do not allow good information sharing characteristics must resort to software control of shared storage and sometimes to the provision of separate copies of program for each process which requires to obey it. We may restate the requirements as follows: a multi-processor system should be able freely to execute code re-entrantly and should be able to access shared information when, and only when, this is a requirement of the program logic. The capability mechanism gives us exactly this property. Information sharing is permitted when required, and entirely denied when access is not logically necessary.

The protection afforded by the capability mechanism is extended in SYSTEM 250 to the interfaces between subroutine linkages. A program can only perform a subroutine call if it possesses the necessary capability for the subroutine. The access condition set into that capability permits 'enter' access only: that is, the capability can only be used to perform a subroutine call and not to gain access to the called subroutine's capabilities and hence to its data structure. Therefore the called subroutine's data structure is completely inaccessible to the calling routine. Similarly, once a routine has performed a subroutine call, the capabilities owned by that routine are denied to the subroutine and this satisfies the converse condition, that the calling routine's data structure is completely inaccessible to the called routine. Information interchange between two such routines is therefore strictly limited to that which the programmer intended: information may be passed as parameters in the form of data and/or capabilities in the machine registers; or information may be made permanently accessible to both calling and called routines, by placing in each routine's data structure a capability pointing to the shared information.

Given the inter-routine capability protection mechanism, it is now possible to construct all programs in a subroutine hierarchy irrespective of whether these programs are conventionally regarded as part of the application software or part of the Operating System software. This fact has had a dramatic effect on the design of the Operating System for SYSTEM 250 because it permits us no longer to regard it as a monolithic software package protected from application software corruption by means of a single impenetratable barrier. Rather, each logical function in the Operating System is treated as a distinct protected subroutine so that the storage protection philosophy within the Operating System structure relies on the same capability mechanisms as those utilised by the application programs. The result of such an organisation is that the system is not split into separate application and Operating System monoliths separated by a 'special supervisor mode' of hardware operation and the distinction between an Operating System and an application subroutine becomes one of administrative significance only.

List Structured Addressing

It has been stated previously that what distinguishes a capability from a conventional base-limit protection mechanism is the ability of the running process to perform load and store operations on capabilities by means of hardware instructions embedded in the program. This contrasts strongly with systems in which the reloading of base-limit registers is undertaken indirectly by software in 'supervisor mode'.

The free copiability property of capabilities enables the programmer to use them as invarient addresses in an arbitrary list structure and, indeed, an unlimited number of copies of a given capability can be generated. This distinguishes the capability mechanism from other invarient address schemes, such as the Burroughs descriptor (reference 5) which essentially restricts the data structure to a tree-like representation.

The arbitrary information sharing properties of the capability are exploited in the SYSTEM 250 Operating System to provide, in a simple manner, multi-programming and multi-access facilities. Firstly, the ability to arbitrarily share code segments means that all Operating System routines can be obeyed re-entrantly by many processes. Secondly, it is possible to strictly protect the information belonging to one multi-access user from that belonging to another. And, thirdly, it is possible for multi-access usersto share information in a controlled manner through a system of directories. The directory structure is similar in concept to that provided by the MULTICS Operating System (reference 6) but it differs in the following important respect: whereas, the directory structure in the MULTICS system is organised as a tree, the directory structure in SYSTEM 250 can be organised as any arbitrary list. Thus, the inter-connection of directories exactly mirrors the hardware level at which the capability mechanism permits an arbitrary interconnection of segments. This feature can be exploited to give precise information sharing properties to a system comprising groups of users of various classes. Our telephone switching example illustrated some of the many man/machine interaction requirements involving the sharing of some information. However, many of the classes of user are performing quite specific and separate tasks which do not require a great deal of administrative interaction. This is reflected in the organisation of directories to which these users are given access: it is the responsibility of the administration to organise the directories into a suitable list structure.

Conclusions

In conclusion, therefore, it has been shown that the requirements of computer systems in telecommunications applications are far removed from the facilities conventionally provided by either telecommunications processors or data processing machines. The facilities of a 'general purpose computer' suitable for these applications have been derived, the main features being incremental expandability of storage and of processing power, automatic reconfiguration of the system following hardware or software failures, and the simple interconnection to distributed telecommunications equipment and to man/machine interface devices.

It has been argued that to satisfy the above features, a computer system should be organised as a multi-processor with each processor equally capable of sharing the work available. This requirement in turn leads to two design conflicts which have been resolved in the design of the SYSTEM 250 computer system by, firstly, the abolition of external interrupts and, secondly, the use of a universal segment identifier known as a capability.

It has further been illustrated that the design solutions to these two conflicts have been turned to our own advantage because the problems involved have forced us to think out from first principles the necessary and sufficient features of a true 'general purpose' computer system. In particular we have been able to avoid an expensive and self-defeating approach to the production of channelling hardware, by recognising that the trend in this area towards increasing complexity implies a trend towards the use of standard processing equipment; we have been able to capitalise on the protection features of the capability mechanism by the design of a modular Operating System organised as a series of protected subroutines; and we have used the concept of free copiability of capabilities to reflect into the user terminal level of the system the idea of an arbitrarily interconnected structure of directories.

In particular, the capability mechanism, which is such a central feature of the SYSTEM 250 hardware architecture, enables us to claim three quite distinct achievements: the protection of information in a multi-processor system against hardware failure, the modularisation of Operating System and application software into a protected subroutine hierarchy, and the efficient and arbitrarily constrained sharing of data structures between competing processes. This leads us to believe that this concept represents a significant and essential advance in both hardware and software technology and that SYSTEM 250 provides both the sufficient and the necessary features of a 'general purpose computer'.

References

1. Graham, R.M. "Protection in an Information Utility", Comm. ACM, 11, 5 (May 1968) pp. 365-369.

2. Dennis, J.B, and E.C. Van Horn "Programming Semantics for Multiprogrammed Computations", Comm. ACM, 9, 3 (March 1966), pp. 143-155.

3. Fabry, R.S. "List Structured Addressing". PhD. dissertation - University of Chicago, Illinois, (June 1970).

4. England, D.M. "Operating System of System 250". Proceedings of the International Switching Conference, Boston, Mas. (June 1972).

5. Burroughs Corporation "The Descriptor" - A Definition of the B5000 Information Processing System - Detroit, 1961.

6. Bensoussan, A, Clingen, C.T, Daley, R.C. "The Multics Virtual Memory". Proc. Second ACM Symp. on Operating Systems Principles, Princeton, N.J. (Oct. 1969).

Acknowledgement

I would like to acknowledge the contribution of my colleagues who were involved in the design of the System 250 and, in particular, the valuable contribution of M. O'Halloran who understood before most of us what Operating Systems were all about. I would also like to thank the Directors of the Plessey Company for permission to publish this paper.

FAULT RESISTANCE AND RECOVERY WITHIN SYSTEM 250

K. J. Hamer-Hodges

Plessey Co. Ltd.
Liverpool, England

Summary

This paper describes some of the aspects of the Plessey SYSTEM 250 real-time processing system, and is an accompanying paper to those presented by my colleagues from Plessey U.K.

The requirements of a Real Time processor system suitable for the control of a communications application are evaluated. The ability of SYSTEM 250 to fulfil these requirements and the hardware architecture which provides the characteristics so urgently required by the communications industry is described.

A general description of the hardware of the processor is included and the use made of capabilities in ensuring the detection and isolation of fault occurences within the working system is described. Particular attention is drawn to the fault recovery sequence and the diagnostic facilities which enable the working system to live through fault conditions and offer the grade of service required by the application.

General Introduction

SYSTEM 250 was designed at the outset to meet the exacting control requirement of telephone or data message switching systems. It should be appreciated that this application demands an exceedingly high standard of performance in almost all of the areas considered important in Real Time applications. Convential computer systems are inadequate when examined against the essential requirements already established by conventional switch equipments. The characteristics of a computer system which will satisfy the stringent requirements of exchange control are summarised under the following headings:

Continuity of service
Ease of expansion
Ability to Evolve
System Partitioning and Security
Flexibility
High Power/Cost Ratio

Continuity of Service

The British Post Office has devised a sliding Scale defining the allowed minimum reliability of telephone exchange control equipment. The scale ranges over steps from failures of the control equipment of less than 15 seconds which can be tolerated up to 50 times per year, to failures of more than ten minutes which should not occur more than once in 50 years. These reliability figures must be maintained despite:

(a) The existence of undetected software errors within the system.

(b) Occasional on-line expansion or modification of both the hardware and the software components.

(c) The need for long periods of unattended operation.

Ease of Expansion

A further requirement is that each individual system should be economically viable from the date of first installation. They must offer a growth potential such that the system is capable of ON-LINE expansion of any facility (e.g. Storage, Processing Power, or Input-Output Capability) by a factor of three during the expected life of 25 years. These extensions should not require alterations or re-compilation of the existing programs or cause any loss of service.

Ability to Evolve

A computer system which is expected to be operational for more than two decades can only remain economic if its architecture permits the inclusion of advances in hardware technology. The software architecture must also provide the flexibility necessary to absorb the undoubted changes which will be required to provide the, as yet, unforeseen facilities to be offered in the future.

System Partitioning and Security

The system hardware and software must be partititioned in a secure manner such that information transfers can be monitored, and faults or errors detected quickly and contained. The aim is to prevent corruption of and/or unauthorised access to system resources, in particular storage media, with minimal overheads in power, cost and complexity.

Flexibility

The control system is required to be flexible in both the hardware and software architecture such that a wide range of configurations with differing requirements can be controlled by differing configurations which minimise the cost of each system. In particular the system must be capable of efficiently controlling large numbers of low activity peripheral devices.

Introduction to System 250

SYSTEM 250 is a modular multi-processor system. The central system modules are Stores, Processors, and Multiplexors. Standard and non-standard Peripheral devices of all types can be attached as will be described subsequently. Twenty four bit word lengths are used for all memory addressing, instruction formats, and data storage. Thus the total memory capacity is in theory in excess of 16 million words. The instruction repertoire has been simplified to twenty seven basic operations, with inter-register, store and register or literal options available when meaningful.

Peripherals Devices are addressed via Control and Data registers which appear to the Processor to be exactly, similar to the normal random access storage connected to the Processors, and it has, therefore, been possible to eliminate all specific peripheral handling instructions. Instead the normal Load Register and Store Register instructions are used, with addresses which specify the appropriate register within the desired peripheral device.

System Architecture

Interconnection of the Processors to attendant storage and peripherals is achieved over a 60 signal bus system, each Processor having an individual bus. Interface Units are used to attach stores and peripherals to these buses. See Fig. 1.

The function of the Interface Unit is to recognise requests for access to the module , resolve contention between individual requests from separate processors, and to allocate each request a cycle of access to the module. A system with up to eight processors is currently possible, with each of the Store Modules equipped with an 8 port Interface Unit. Peripheral Devices, however, are equipped with only 2 port Interface Units. When, therefore, there are 3 or more processors, peripherals connect to a Peripheral Bus system, driven by Multiplexors which can be equipped with 8 ports. Thus the more expensive 8 port Interface Units are not required throughout the Peripheral area. Two Multiplexors are required for security, and if either one should fail all traffic is passed through the alternative unit. See Fig. 2.

Up to 40 Modules can be attached to each Bus, over distances of 100 metres.

Only high activity, or fast speed devices, need be connected directly to the Bus system, e.g. Backing store devices. Low activity or slow speed devices such as user terminals or the application terminals of a real time system are connected to a serial data collection and distribution system known as the Serial Medium. See Fig. 3.

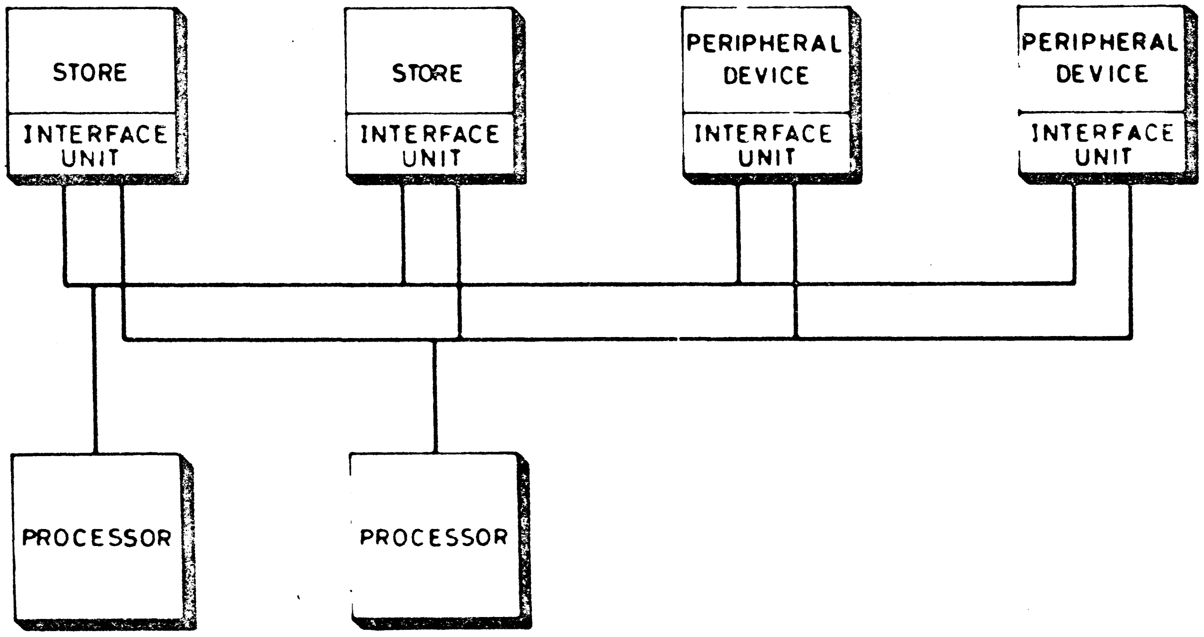

Fig 1 Typical two Processor system

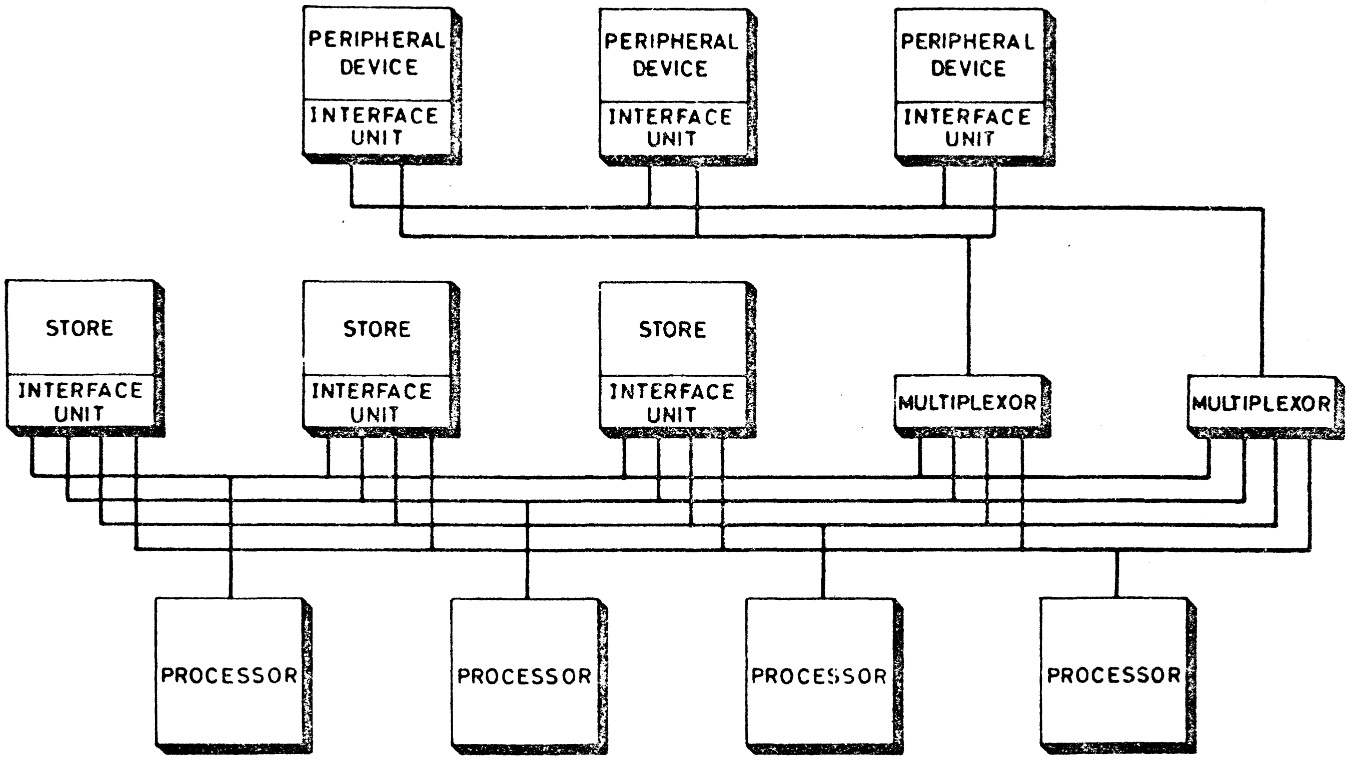

Fig 2 Expanded Processor system showing Multiplexors

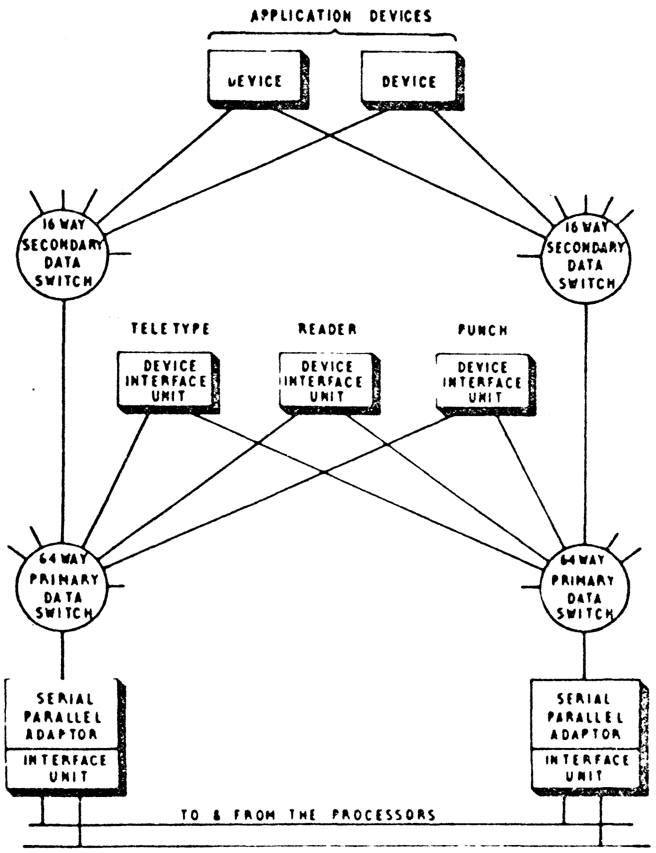

Fig 3 The Serial Medium

The Serial Medium is controlled by a device connected to the Bus system and known as the Serial Parallel Adapter (SPA). Packets of address and data are collected or distributed by the SPA, via a cascaded arrangement of Data Switches which multiplex and demultiplex the message paths. Terminals can be connected to the Serial Medium at any switch outlet so that some devices may be connected at the first switch others at subsequent switches. Each device has a unique address which is used to route outgoing messages to the device, and is assembled during incoming messages to the SPA. Check codes are used to validate all message transfers.

Each device which is connected to the Serial Medium is equipped with a 2 port Serial Access Unit for connection to two separate Serial Mediums. This is done for security of communication if either path should fail.

The modular structure of the SYSTEM 250 has been arranged so that individual system parameters can be matched in the most economic way possible, Stores for example can be built up in units of 8, 16 or 32K in slow, medium or fast access times ranging from 1μs to 300ns to match the data storage requirements. Numbers of processors can likewise be matched to the work requirements and the security requirements. The number of peripheral terminals can similarly be equated to the requirements of each installation. Further the System can be expanded in small steps by the addition only of the required module.

Capabilities

Each Processor has access to all modules connected to the system. Consequently each Processor represents a security hazard if either a hardware fault or a software error should corrupt a location by accident. The concept of Capabilities has therefore been implemented in the Processors to protect against corruption of invalid areas of storage, including Peripheral devices. Reference 1 discusses the necessity of capabilities and provides more detailed references. Capabilities are descriptors which identify the separate 'logical' entities within the system and the users access rights to the logical block. The Operating System loads these logical blocks into physical address space and allocates the Base and Limit address values accordingly via a map known as the System Capability Table.

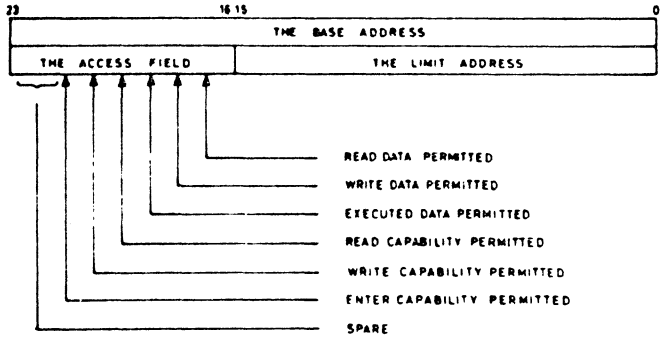

Fig 4 The Capability format

The Hardware of the Processor provides eight Capability Registers into which can be loaded the Base, Limit, and Access field of separate addressable blocks required by the program. It is emphasised that the use of capabilities in no way restricts the flexibility of programming at writing time. The function of capabilities is to ensure that once defined by the program, the limits (Base, Limit and Access Rights) are observed by the hardware and the process at run time, even under fault conditions.

Thus capabilities are a valuable mechanism in protecting against the type of fault which causes the progressive corruption and final breakdown in a multi-processor system. The basic aim in using capabilities is to restrict the effect of a fault to the currently running process, and to identify the existence of a fault immediately it occurs.

Processor Architecture

In order to understand the principles of system operation it is necessary to describe the architecture of the Processor . Reference 2 describes many of the hardware aspects of the processor which are not described in detail here.

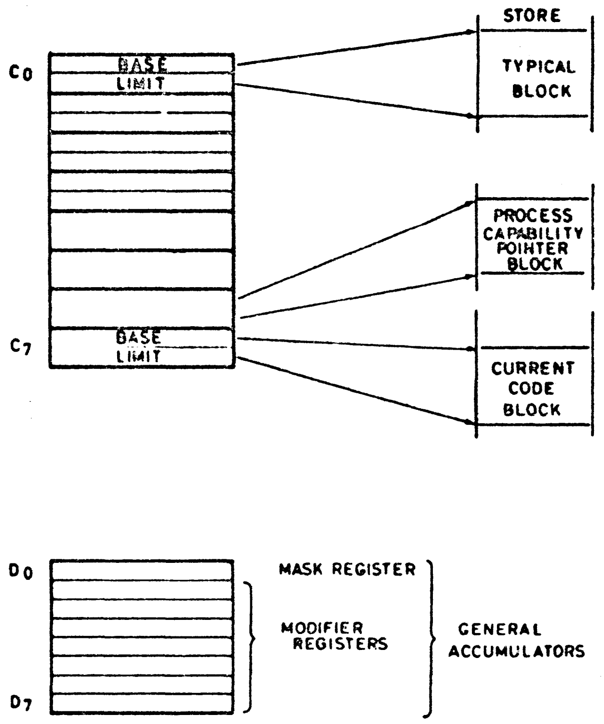

Fig 5 The working Register set

There are eight fifty bit (48 + 2 parity) general purpose capability registers C0-C7. Conventionally C7 is required to hold the currently executed code block and C6 defines the Process Capability pointer block, which in general defines the working set of 'capabilities' available to the code block in execution. The remaining registers C0-C5 are loaded by a standard instruction, 'Load Capability', under programmer direction.

There are eight twenty four bit data registers D0-D7 all of which can be used as accumulators, and seven of which can be used as address modifiers. In addition to these two sets of working register the hardware provides further 'hidden' Data and Capability registers required for efficient operation, and these cover timer registers, indicators, etc.

All memory addressing is performed by the addition of a selected Base value of capability register to an offset (derived from the instruction). Before any store operation is performed this final memory location is then checked to be greater than the Base value, since negative modification is possible, and less than the Limit value. Similarly, the micro-program action, Read or Write, is checked to be permitted by the Access field of the selected capability before the Store operation is allowed to complete. Fault interrupts are generated if any violation of the capability is attempted.

Clearly the system places great reliance upon the validity of the capability registers and the data held by them. Therefore a considerable number of checks are involved when loading and using capability registers which together ensure that no single hardware or software failure can pass undetected by one or other of the checking mechanisms. These mechanisms include a twenty four bit sum check, parity checks and register addressing checks.

Six basic capability manipulation instructions are provided which permit the programmer to 'Load' capability registers, 'Pass' capability blocks from one procedure to another, 'Call' and 'Return' from sub-routines, and 'Changing Process'. In all these cases, however, the Base, Limit, Adcess Field values of the capabilities manipulated are set by the operating system and not directly under the programmers control.

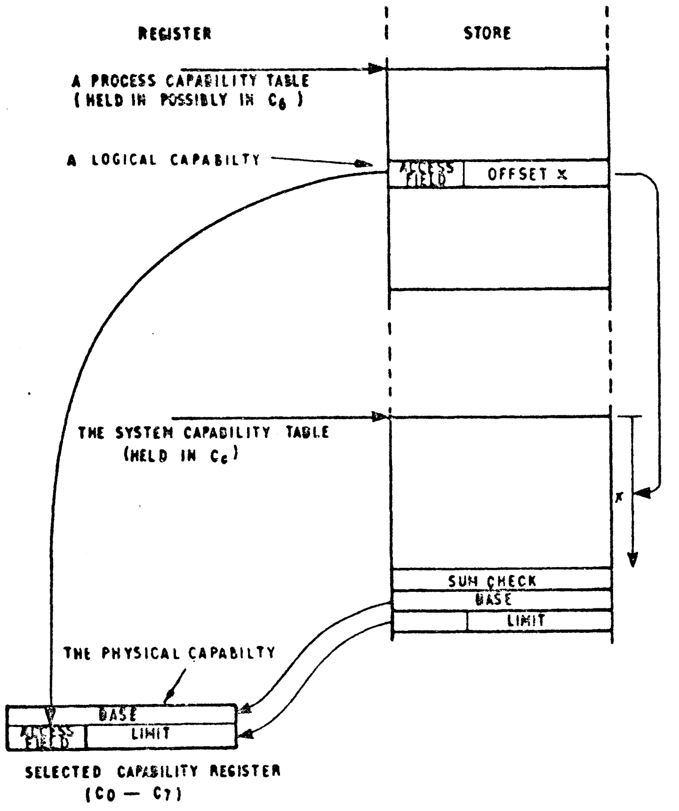
REGISTER
STORE
A PROCESS CAPABILITY TABLE (HELD IN POSSIBLY IN C_6)
A LOGICAL CAPABILTY
ACCESS FIELD
OFFSET X
THE SYSTEM CAPABILITY TABLE (HELD IN C_6)
X
SUM CHECK
BASE
LIMIT
THE PHYSICAL CAPABILTY
BASE
ACCESS FIELD
LIMIT
SELECTED CAPABILITY REGISTER (C0 — C7)

Fig.6 The Load capability sequence

The selection of the Base, Limit and Access Field is arranged via the System Capability Table held in one of the 'hidden' capability registers. Within this 'map' is described all the currently available blocks referenced in main store. Each user of the system, has a set of 'capabilities'. Each capability specifies an Access Field and the offset of one of the 3 word packets held in the 'map'. The Capability manipulation instructions reference the available 'capabilities', this in turn enables the hardware to select and load the assigned Base, Limit and Access Fields into the Capability registers of the Processor. Thus the logical capability is converted into a physical address at run time.

It must be stressed that although the programmer is at liberty to load into a register the assigned Base and Limit values of any of his available capabilities he cannot alter either the values of his own set of logical capabilities, or the corresponding physical Base and Limit values. This is effected by disallowing the WRITE DATA facility on a capability block.

Therefore at all times, the range of Memory locations which can be accessed is limited to the available 'capabilities' and the corresponding Base Limit values held in the System Capability Table.

Fault Detection and Recovery

In order to protect the working system from progressive collapse due to the migration of faults through the system, the Processor performs a Fault Interrupt immediately the fault condition is recognised, and before any actual capability violation can occur. The Fault Interrupt sequence is critical in order to preserve the system security and therefore in understanding the system recovery mechanism. The hardware sequence is consequently described in some depth, Reference 3 elaborates the system philosophy, and recovery sequence ensuing after a hardware fault.

The actions executed by the microprogram are repeatable and subsequent fault indications cause the sequence to be re-attempted.

FAULT ENTRY

↓

1 NULLIFY ALL CAPABILITY REGISTERS EXCEPT START UP BLOCK

↓

2 PRESERVE FAULT INDICATORS

↓

3 INCREMENT START UP AREA TO NEXT MODULE

↓

4 RELOAD SYSTEM CAPABILITY TABLE FOR START UP SEQUENCE

↓

5 "CHANGE PROCESS" TO START UP PROCESS

Fig. 7 The Fault Interrupt Sequence

Firstly the currently loaded Base and Limit values are corrupted to give invalid parity detection. This ensures that even given incorrect sequencing through the Fault Interrupt microprogram any attempts to access the memory locations of the previously running Process are prevented.

Secondly the Fault Indicator register is stored into a hidden register to preserve the fault indication. The Fault Indicator register is then cleared.

Thirdly, the Start Up Capability Register C(S) is incremented by 2^{16} so that it now references a different memory module. This ensures that during multiple fault conditions the Processor attempts to 'Start Up' from each of the available store modules in turn until it succeeds.

Fourthly, the Capability Register referencing the System Capability Table C(C) is reloaded with a Sumcheck, Base and Limit value held as the first three entries defined by the Start Up Capability register C(S). The block thus loaded references a new and limited set of Base and Limit values available to the Start Up Process.

Finally the Change Process microsequence is attempted using the Capability held as the fourth entry in C(S).

When each of these steps has been executed successfully the Fault Interrupt Process is activated. This Process will run with a limited set of memory locations available thus preventing interference with other Fault Free Processors.

The pre-requisite of the Fault Interrupt sequence is that at least one valid copy of the Start Up Block and the associated Program and Data block exists in any one of the equipped Store modules. Similarly if the Processor has a hardware failure which prevents the successful activation or completion of the Start up Process the hardware is condemned to an eternal cycling of the Fault sequence in an endless attempt to recover.

Note that the system recovery sequence which follows a fault detection can be made as rigorous as the application requires, Reference 3 discusses this in more detail.

On-Line Diagnostic Facilities

In order to achieve high reliability at reasonable cost the Mean Time to Repair faulty modules must be reduced to a minimum. In broad terms this has two effects. Firstly, the possibility of a second failure within the critical part of the system during the 'down time' of the first module is minimised, thus improving the system reliability, or alternatively, for a fixed reliability the number of redundant modules of any one type is minimised thus reducing system cost.

SYSTEM 250's diagnostic software and maintenance procedure is an integrated system which minimises the system repair time. The novel aspect of this system is concerned with Processor diagnostic software.

Processor diagnostics are normally an extension of functional test programs. They are run on suspect machines in the hope that the fault will not be serious enough to prevent the successful completion of the test program. Output is then produced which indicates the faulty component. There are two hazards in this approach, the first is that the fault could reside in the 'hard core' of the machine and either prevent the successful output of any message, or faulty output may be obtained, second, the processor, although suspect, requires the use of system resources in order to run and output any message.

For System 250 this is unacceptable for two crucial reasons. Firstly, the whole nature of the design is oriented towards a 'hard core' which includes the whole machine, it is in this way that faults are indicated immediately. Secondly, faulty processors are trapped in the fault recovery sequence deliberately so that they cannot make use of systems resources.

However, as a consequence of System 250's multiprocessor philosophy, it has been arranged that the diagnostic routines run on a working processor which then interrogates the suspect machine.

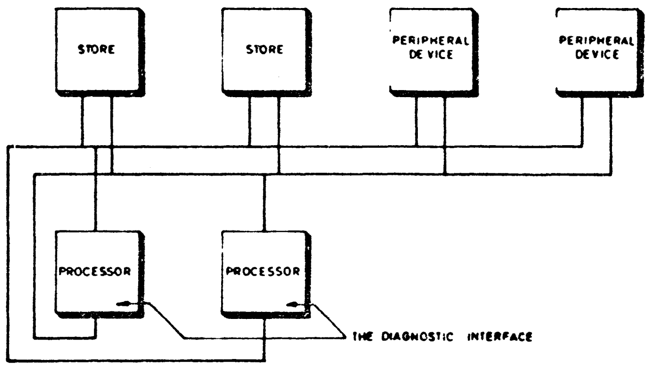

Fig 8 The connections of the Processor Diagnostic Interface

Each Processor module has an optional 'Diagnostic Interface'. This interface is exactly the same in operation as the Store and Peripheral Interfaces connected to the Store Bus. Each Processor can therefore be connected to the Test Interface of one of the other machines in the System, either directly or via a Multiplexor. The internal logic of each processor is therefore addressed as memory locations. The appropriate 'Capabilities' must be loaded into the hardware registers of the interrogating processor in order to address the suspect machine.

A set of Commands are provided as part of the Diagnostic Interface which facilitate the operation of certain essential functions.

STOP MAIN PROCESSOR CLOCK

START MAIN PROCESSOR CLOCK

PERFORM SINGLE SLOT WORKING

PERFORM SINGLE INSTRUCTION WORKING

REPEAT A PARTICULAR INSTRUCTION

STOP AT A PARTICULAR INSTRUCTION ADDRESS

STOP AFTER "n" SLOTS

STOP AT A PARTICULAR MICROPROGRAM SLOT

STOP AT A FAULT CONDITION

MONITOR MICROPROGRAM CONTROL SIGNALS

FORCE MICROPROGRAM CONTROL SIGNALS

MONITOR INTERNAL REGISTERS

FORCE INTERNAL REGISTERS

Fig.9 The Diagnostic Interface Commands

In the simplest terms the registers can be loaded with a known pattern, clocking functions can be performed and the register can be examined and compared with a known result. Discrepancies are isolated to single paths and the results indicate far greater fault resolution than is possible by traditional methods.

The diagnostic package will provide fault analysis down to one board (or a small number of boards when, for example, 'wire-or' functions are faulty).

Conclusions

Each characteristic of SYSTEM 250 was conceived to satisfy one or more of the design requirements detailed at the start of th's paper.

A SYSTEM 250 CHARACTERISTIC	THE REQUIREMENT
CAPABILITY PROTECTION MODULARITY REDUNDANCY MULTIPROCESSOR TRAFFIC SHARING FAULT DETECTION AND RECOVERY ON-LINE DIAGNOSTICS	CONTINUITY OF SERVICE
STANDARD HARDWARE INTERFACES STANDARD SOFTWARE INTERFACES MODULARITY MULTIPROCESSOR DATA COLLECTION AND DISTRIBUTION	EASE OF EXPANSION, ABILITY TO EVOLVE, FLEXABILITY & POWER/COST RATIO
CAPABILITY PROTECTION STANDARD HARDWARE INTERFACES STANDARD SOFTWARE INTERFACES	SYSTEM PARTITIONING & SECURITY

Fig.10 The System Characteristics

While not exhaustive, it is hoped that this paper, in conjunction with the others presented by my colleagues, has indicated the principles of operation of SYSTEM 250, its architecture, and its power.

References

1. D.C. Cosserat - 'A Capability Oriented Multi-Processor System for Real-Time Application' presented at this Conference.

2. D. Halton - 'Hardware of SYSTEM 250 for Communication Control' Proceedings of the International Switching Conference, Boston, Mass. June 1972.

3. C.S. Repton - 'Reliability Assurance for SYSTEM 250 a reliable, Real-Time Control System' presented at this Conference.

Acknowledgement

I would like to thank the many colleagues on whose work this paper is based and the Directors of the Plessey Company for permission to publish it.

RELIABILITY ASSURANCE FOR SYSTEM 250
A RELIABLE, REAL-TIME CONTROL SYSTEM

C. S. Repton

Plessey Co. Ltd.
Liverpool, England

Summary

System 250 is a multi-processor system designed for real-time communication applications where very reliable operation is required. The initial application of this system (control of a telephone exchange) is required to achieve a mean time between failure of 50 years, where a failure is defined as a system outage lasting over ten minutes.

The paper describes in a general way the problems involved in providing this degree of reliability, and some solutions which can be adopted. The approach which is being used in the design of System 250 is described.

Particular emphasis is placed on the initial stages of recovery which ensure that a fault-free system configuration is set up and that a basic minimum set of programs are correctly loaded and working, allowing the system to bootstrap its way back into full operation. The hardware and software mechanisms used to achieve this basic level of recovery are described in some detail, and the methods used to secure these mechanisms themselves against the effect of fault conditions are also considered.

Introduction

The application of computer systems to real-time control situations is rapidly expanding. Many of these applications, such as air traffic control and communication systems, are essentially continuous activities which demand very reliable control systems. This means that the design of highly reliable computer systems is becoming increasingly important. This paper describes the methods used to secure a real-time, multi-processor system (System 250) against failure and discusses some of the problems involved in providing reliable system operation.

System 250 has been designed for communication applications, such as control of telephone switching, where continuous, reliable operation is required. A typical requirement of this type of application is a mean time between system failure of 50 years, where a system failure is defined as an outage lasting over ten minutes.

Previous papers[1,2] have outlined the overall configuration of System 250. The main features are that the system uses a group of functionally identical processor units connected to a group of identical store units. This type of configuration can be made to perform like one large, very powerful computer, and yet its power can be economically increased in small steps simply by adding more processor or storage units. Since all units are functionally identical any store module can replace any other store module, and similarly any processor unit can replace any other processor. This means that equipment failures can be catered for fairly simply. In the event of a unit failure the faulty unit is isolated and the functions of that unit are then reallocated to other modules in the system which have some spare capacity.

The software which is used to control this hardware configuration can usefully be considered as a number of distinct layers or levels[3]. As each new layer of software is added to the system it is used to extend, or present in a more convenient form, the facilities which are available. In effect the first layer takes the bare facilities provided by the machine instruction set and adds to them by providing further facilities within the software. This provides subsequent levels in the hierarchy with an enhanced version of the original machine, a kind of 'virtual machine'. The additional levels use this extended machine to produce further, more powerful facilities. Thus as one progresses along the hierarchy the facilities provided by the virtual machine at each level become increasingly useful and powerful.

In the case of System 250 the first software level takes the multi-processor, multi-store system and converts it into a virtual machine which appears to subsequent levels to be one large, very fast processor with one large store. All the problems associated with multiple processor operation are handled at this initial level, and subsequent layers need not consider the multi-processor nature of the system. The next level in the hierarchy provides convenient input/output facilities and controls the backing store devices such as discs, so that lower layers see a very much larger store system than that provided by the main store alone. The next level provides operator communication and facilities such as program assembly, editing, job control etc. Finally on the last level come the application programs which actually perform the real-time operations (Fig. 1).

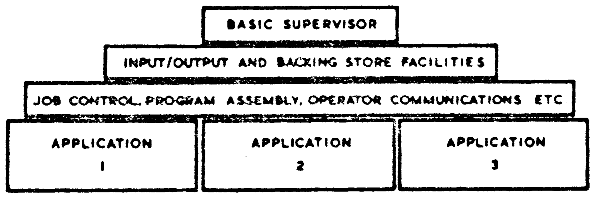
BASIC SUPERVISOR
INPUT/OUTPUT AND BACKING STORE FACILITIES
JOB CONTROL, PROGRAM ASSEMBLY, OPERATOR COMMUNICATIONS ETC
APPLICATION 1
APPLICATION 2
APPLICATION 3

Fig. 1
Software Structure of System 250

There may be several sets of application programs in a system such as this. For example, one central control system may control several remote telephone exchanges. Other functions may be required which are related to, but not part of, the main real-time activity. For example, a maintenance sub-system to allow on-line testing may be added or a program development sub-system to allow new programs to be developed and debugged before being introduced into the real-time system.

Recovery Mechanisms

Based on this broad description of System 250 let us now consider the type of facilities and mechanisms which will have to be built into the system to allow it to recover automatically from fault situations.

Obviously the system will have to cope with failures within individual processor units and store modules, so that we require some means of detecting that a fault has occurred and locating the fault to a particular module. The faulty module can then be isolated so that it cannot interfere with the rest of the system. Finally any data which may have been lost or corrupted by the fault must be restored so that normal operation can continue. Typically this will involve reloading lost programs and data in the event of a store fault, and abandoning or reconstituting suspect data after a processor failure.

The system will also have to deal with software faults. On the basis of past experience it seems inevitable that even after thorough testing and commissioning all but the smallest system will still contain design errors in the software. This means that the system will occasionally behave unpredictably when certain, rather rare, combinations of data or timing circumstances occur. All that is required in this case is to reset any data which has been affected by the failure and restart processing using fresh data. This type of data recovery mechanism is similar to that required to deal with the after-effect of processor failures, as described above.

Thus, in general, each recovery action includes three distinct phases:- The first is the detection that an error has occurred. The second is an attempt to locate the fault to a particular hardware unit. This may not succeed, either because insufficient information is available, or because the fault is caused by a software problem. Finally the third phase will involve some form of data recovery or restart procedure which will allow the system to resume normal processing.

Within System 250 the mechanisms used at each stage of recovery are as follows:-

The error detection mechanisms which are used are:-

(1) fault detection circuits built into the hardware.

(2) software consistency checks and time-outs to monitor overall system performance.

(3) test routines run in background mode.

The methods used to locate the fault to a particular unit are:

(1) Persistent fault conditions reported by check circuits.

(2) If the error detection mechanism implicates a particular unit or units (for example hardware check circuits or test routines) a fault count associated with the unit or units can be incremented in order to detect persistently failing devices.

(3) A localised test procedure can be used to test units which are suspect as the result of an error indication from a hardware check circuit or failed test routine.

(4) The testing sequence can be extended to cover all units within the control system.

(5) As a last resort units can be switched out of system on a trial basis in an attempt to find a viable system configuration.

There is obviously a very wide range of data recovery and restart procedures which can be adopted. We have found it useful to adopt three stages of recovery action which provide progressively more extensive restart facilities. These are:-

(1) Process Restart Each process, or transaction, in the system has a defined recovery action which can be activated if that process meets any form of error condition. The recovery action involved will vary depending on the nature of the transaction, and these can range from regenerating data areas, and restarting the failed process in the case of a vital system function, such as a disc handler, to simply ending the failed process and printing diagnostics.

(2) Area Restart Each functional area within the system has a defined recovery action which will allow read/write data to be regenerated from duplicate files held on disc by that area. This may allow complete data regeneration, but more usually, some transactions will be abandoned and only the most important functions will be made restartable by storing redundant information on disc. This type of restart is commonly referred to as a 'warm start'.

(3) Area Reload Each functional area also has a defined recovery action which will allow processing to be restarted from read-only information in duplicate, sum-checked files held on write-protected areas of the backing store. This form of recovery obviously involves abandoning all current transactions, reinitialising the system and then resuming processing new

transactions. This type of restart is commonly known as a 'cold start'.

Recovery Procedures

We have now considered the basic elements which are available for use in constructing the required recovery procedures. Before moving on to discuss the form taken by these recovery procedures it is worth making the following observations:-

(1) the hardware test and data recovery procedures involved can themselves disrupt system operation, for example it is difficult to perform a complete test on every hardware unit in the system without causing some disturbance to normal on-line processing, and the various data recovery procedures often abandon perfectly valid transactions rather than attempt a complex validity checking operation.

(2) the error indications do not always pinpoint the source of the fault or the identity of the corrupt data. Processors may trigger hardware check circuits as the result of attempting to process invalid data corrupted by faults elsewhere in the system, and it is impossible to predict just how much data may have been disturbed by any given software fault.

This means that it is very difficult to adjust the recovery action so that the fault is corrected and yet the disturbance to system operation is minimised.

In the circumstances the best strategy is to combine the various fault location and data recovery/restart procedures into a sequence of recovery actions. Initially the action which causes least disruption to system operation is used. If this fails to clear the fault, as indicated by further error reports, then increasingly powerful (and hence more disruptive) recovery actions are used until the fault is cleared, as indicated by the absence of further error indications.

The sequence of actions which has been adopted in System 250 is shown in Fig. 2. Error indications which do imply the location of a fault (hardware check circuits and failed test routines) cause a fault count to be incremented for the unit, or units involved. If one unit is consistently implicated then the fault count indicates this. A local testing procedure for the suspect units is also activated. If either of these mechanisms detect a consistent fault the system is reconfigured to isolate the faulty unit. In the case of a hardware check circuit indication it is also necessary to restart the process which was running at the time of failure as the data associated with this transaction is now suspect. Repetitive errors detected by hardware check circuits within a short time interval suggest that the fault may be due to a software problem within the failing area rather than a hardware fault. Therefore in this case the

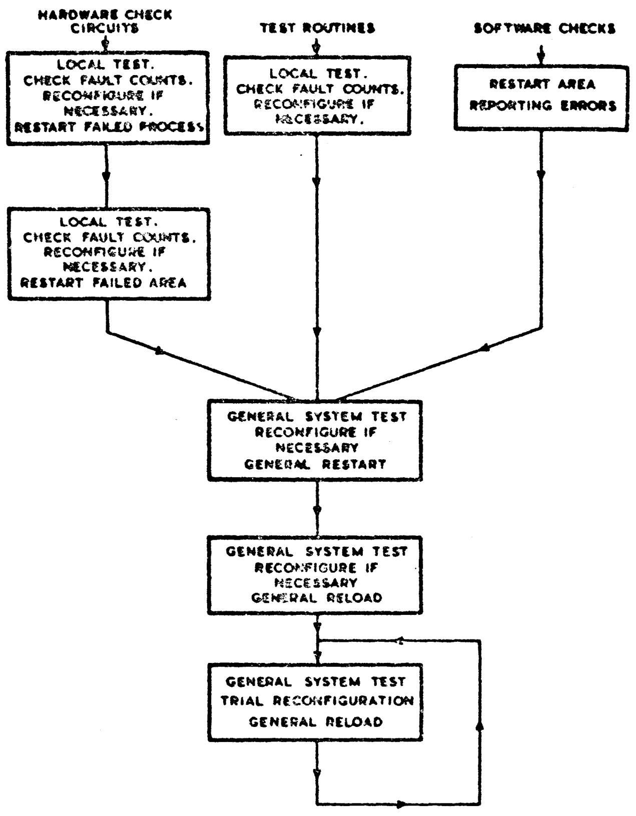
HARDWARE CHECK CIRCUITS
TEST ROUTINES
SOFTWARE CHECKS
LOCAL TEST. CHECK FAULT COUNTS. RECONFIGURE IF NECESSARY. RESTART FAILED PROCESS
LOCAL TEST. CHECK FAULT COUNTS. RECONFIGURE IF NECESSARY.
RESTART AREA REPORTING ERRORS
LOCAL TEST. CHECK FAULT COUNTS. RECONFIGURE IF NECESSARY. RESTART FAILED AREA
GENERAL SYSTEM TEST RECONFIGURE IF NECESSARY GENERAL RESTART
GENERAL SYSTEM TEST RECONFIGURE IF NECESSARY GENERAL RELOAD
GENERAL SYSTEM TEST TRIAL RECONFIGURATION GENERAL RELOAD

Fig. 2
Sequence of Recovery actions used by System 250

recovery action is extended to cover the failing area rather than just the process involved.

Faults detected by software checks cause a restart of the functional area detecting the fault. If the error is due to a software problem within that area this should clear the fault.

Further repeated error indications of any kind cause a general system test to be performed which thoroughly tests all control system elements. Any faulty units are isolated and the system is restarted by means of an Area Restart applied to all functional areas. This procedure will eliminate any data corruption in main store and will recove the vast majority of all solid hardware faults.

If further error indications are generated then another general system test is initiated in the hope of detecting possible intermittent hardware faults. Any faulty units are isolated and the system is reloaded by means of an Area Reload applied to all functional areas. This will reload the system using duplicate read-only files from backing store. This eliminates any possibility that further system failures can be caused by corrupted data generated by an earlier fault.

After this stage the only faults which can remain undetected are intermittent hardware faults or solid faults not detected by the test routines. Therefore, as a last resort, subsequent fault reports initiate another general system test in a further attempt to detect intermittent failures.

If no new faults are found one of the units is switched out on a trial basis (trial reconfiguration) The system is then reloaded by applying an Area Reload to all functional areas. Repetitive application of this procedure will eventually eliminate faulty units which remain undetected by the test routines.

Overall Structure of the Security System

The previous section discussed the sequence of actions which should be followed when an error is detected within the central control system. The group of programs concerned with controlling this sequence are referred to as the basic recovery system, and form an additional layer in the software hierarchy (Fig. 3).

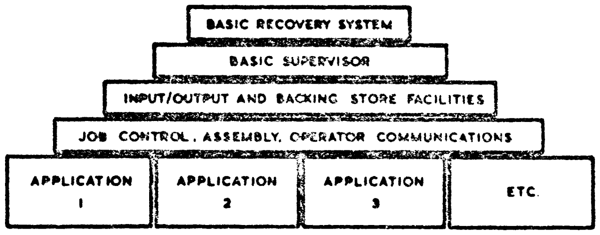
BASIC RECOVERY SYSTEM
BASIC SUPERVISOR
INPUT/OUTPUT AND BACKING STORE FACILITIES
JOB CONTROL, ASSEMBLY, OPERATOR COMMUNICATIONS
APPLICATION 1
APPLICATION 2
APPLICATION 3
ETC.

Fig. 3
Software Structure showing Basic Recovery System

When discussing the functions provided by the various levels in the hierarchy it was shown how the basic supervisor, which contains the scheduling and store allocations routines, effectively concealed the multi-processor, multistore nature of the system from the lower levels. Programs involved in lower levels could be written on the assumption that they would run on one large processor with one large store. The basic recovery system performs a similar function in that processor and store failures are dealt with at this level, and lower levels in the hierarchy do not need to be concerned with the possibility of hardware failures. They can be written on the assumption that they are always held in a fault-free store module, and are obeyed by a fault-free processor. Thus although several copies of the basic recovery procedures must be available to protect this level against store failures, programs on lower levels do not need to be duplicated. If a store module fails, the programs held in that module will be reloaded into a new module by the basic recovery system. Therefore, placing the basic recovery system at the highest level in the hierarchy reduces to a minimum the amount of program which must be replicated. It also simplifies the system since lower levels do not need to consider the possibility of hardware faults.

The software checks required to provide an error detection mechanism should be distributed throughout the system so that each level contains its own independent set of checks. Similarly it is convenient to provide data recovery and restart procedures on a per level basis. This means that each level becomes an independent functional

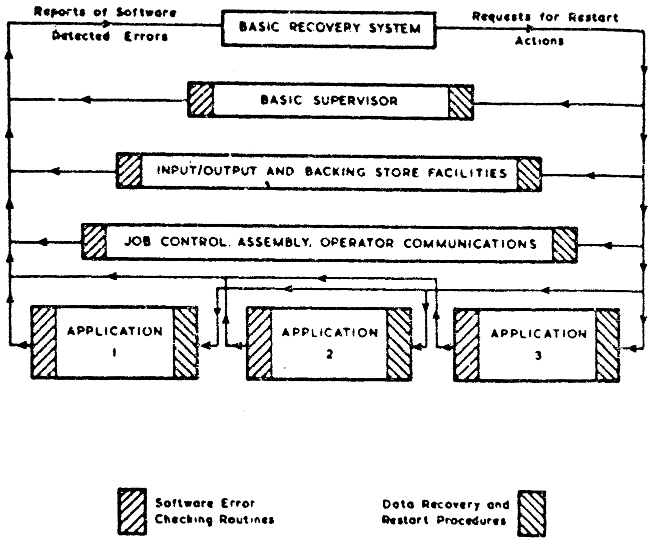
Reports of Software Detected Errors
BASIC RECOVERY SYSTEM
Requests for Restart Actions
BASIC SUPERVISOR
INPUT/OUTPUT AND BACKING STORE FACILITIES
JOB CONTROL, ASSEMBLY, OPERATOR COMMUNICATIONS
APPLICATION 1
APPLICATION 2
APPLICATION 3
Software Error Checking Routines
Data Recovery and Restart Procedures

Fig. 4
Software structure showing Basic Recovery System & Communication Paths to the Rest of the System

area, with its own set of software checks, and its own restart procedures.

The software checks report any errors to the basic recovery system which can then initiate the appropriate recovery action, which may involve invoking restart procedures provided by the lower levels. This modified hierarchical diagram is shown in Fig. 4.

This type of system structure means that as one progresses down the hierarchy not only do the number of facilities available increase, but it is also possible to make wider assumptions about the state of health of the system. Below the basic recovery system programs may be written on the assumption that all hardware faults have been eliminated from the system. The only responsibility that these lower levels have with respect to system reliability is to maintain an overall measure of performance through the software checks on that level, to report consistent faults to the basic recovery system on the assumption that the degradation is due to some form of system fault, and to provide the standard recovery procedures. Below the level of the basic supervisor it may also be assumed that reliable store allocation, and scheduling facilities are available, since it is the responsibility of the software checks and restart procedures within the basic supervisor to ensure this. Below the input/output level it may also be assumed that reliable system peripherals are available, and, for example, an application program written to test a particular piece of application hardware can ignore possible side effects due to faults on the input/output channels. It is the responsibility of the input/output routines within the operating system to eliminate these faults. This expanding level of confidence continues right down to the application/operating system interface where it may be assumed that processors, stores, input/output c'annels and system peripherals are working correctly and that the full range of operating system facilities is available. Of course, it is the responsibility of the application programs to cover the effects of faults in any special peripherals controlled - wholly by that application.

Thus the overall reliability of the system is based on a hierarchy of guarantees. At the top of the hierarchy the basic recovery system provides fault-free stores and processors. Working from this base the other levels can then guarantee fault-free input/output devices and operating facilities to the application programs. By using this wider base the application programs can now secure their own specialised peripherals against failure. In many ways this hierarchy of guarantees parallels the functional build-up of the system, which is based on using the facilities provided by higher levels to make extensive or sophisticated facilities available to lower levels.

Securing the Security System

In the scheme outlined above everything depends on the ability of the basic recovery system to guarantee fault-free processors and stores to the lower levels. One of the main problems involved in producing a workable security system is to ensure that the basic recovery system itself is not disabled by fault conditions. Obviously several copies of these recovery programs must be provided in seperate store modules to protect them against store failure, and some form of protection must be provided to prevent these multiple copies being overwritten by a faulty processor. The recovery programs must also be accessible to several processor modules, to cover processor failures.

These requirements could be most easily met by nominating some, or all, of the processors as 'fault handling' units and providing each with a private store module containing a copy of the recovery programs, Fig. 5. In the event of a store or processor failure one, at most, of the store/processor pairs would be disabled and unable to take effective action. The other processors would then be able to clear the fault and recovery system operation.

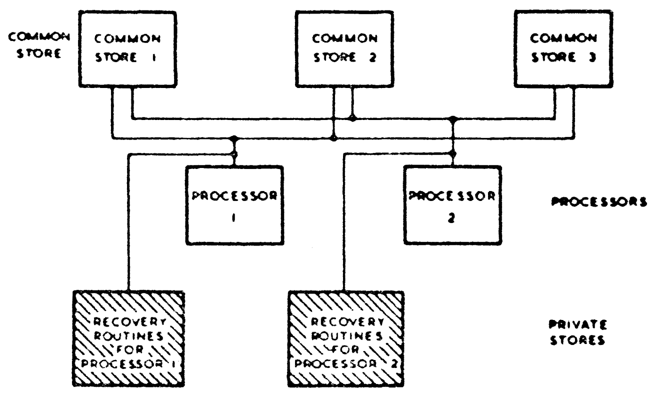

Fig. 5
System Diagram illustrating 'Store per fault-handling processor' approach

As each of the store modules containing the recover programs would be accessible to one processor only, this would protect the recovery programs from faults in other processors.

However, this method does have considerable disadvantages. Because each of the fault handling processors accesses a particular copy of the recovery programs when a fault is detected it is difficult to prevent faults in the store associated with these programs also disabling the processor. This effect considerably reduces the mean time to failure of the processors. In addition this scheme can involve a considerable cost penalty, particularly in large systems, because a store module per 'fault handling' processor is required for the recovery programs.

In contrast System 250 allows any processor access to all copies of the recovery programs (see Fig. 6). This means that:-

(1) failure of a store containing a copy of the recovery programs does not also disable a processor.

(2) it is only necessary to provide sufficient copies of the recovery programs to protect the system against simultaneous store failures.

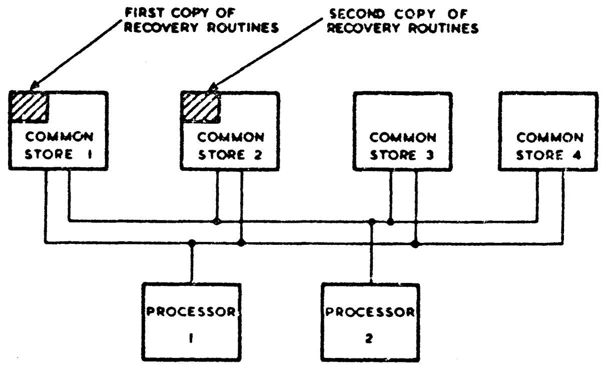

Fig. 6
System diagram illustrating System 250 Approach to Fault Handling

This arrangement is made possible by two features of the processor hardware:-

(1) the capability mechanism, which was described in a previous paper[1], provides a very secure store protection facility, and protects the recovery programs against the possibility of being over-written in the event of a hardware or software fault.

(2) the fault interrupt mechanism, also described in a previous paper[1], which together with the test program in the first section of the recovery programs, is used to control access to the recovery programs.

To illustrate this scheme assume, for the moment, that the only form of entry to the recovery programs is via a fault interrupt. This may be an involuntary interrupt resulting from an attempt to perform some illegal operation, or it may be a deliberate attempt to invoke the recovery mechanism because some error condition has been detected by the software. On taking a fault interrupt the PP250 hardware first disables all the current capabilities held in the machine, thus preventing further access to store. It then attempts to reload a new set of capabilities from a pre-designated location in store. If this is completed successfully the resulting capabilities are used to access the first part of the recovery program. This is a test program which is arranged as a maze. The only possible exit from this maze is via a further capability which is created bit by bit as the machine proceeds through a series of tests. These tests are designed to completely check the hardware and the 'read only' blocks (programs and data) associated with the recovery program. If an error is detected at any stage then another fault interrupt is forced. This causes the processor to reattempt the capability load from the next available store module (see Fig. 7).

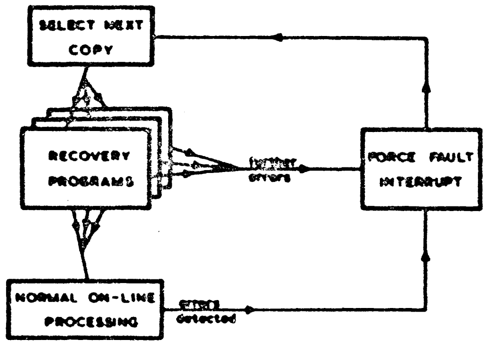

Fig. 7
Security System using fault interrupt mechanism & Replicated Recovery programs

This mechanism provides a dual function. First it ensures that a faulty processor is constrained to endlessly cycle through the storage system in an attempt to find a test program which it can obey successfully. The only capabilities available to the faulty machine at this time are associated with the test program, so that it is unable to interfere in any way with the operation of the on-line system. Secondly it allows fault-free processors to search through the storage system to find an uncorrupted version of the recovery programs.

The mechanism described above, although considerably better than the 'copy per processor' method, does have some disadvantages. The first is that before any recovery action can be taken the processor involved must obey a lengthy (100-200 msec) test program. The second is that all the recovery programs must be replicated. Both of these precautions are unnecessary in some fault situations where the fault is unlikely to disable the on-line system in any way, and the recovery action is fairly simple. For example, software faults which corrupt data within application programs are unlikely to affect the normal running of other programs. Once the fault condition is detected it is only necessary to activate the data recovery/restart routines for the particular application to recover system operation.

This rather minor kind of fault can be dealt with quite adequately by programs which exist in the on-line system and run in the normal way. However these programs do need some form of protection so that if they themselves are disabled by the fault, or are unable to cope with the fault situation in some other way, then the more powerful, replicated programs can be activated. Thus some form of monitor mechanism which can detect the failure of these unreplicated programs is required, as shown in Fig. 8.

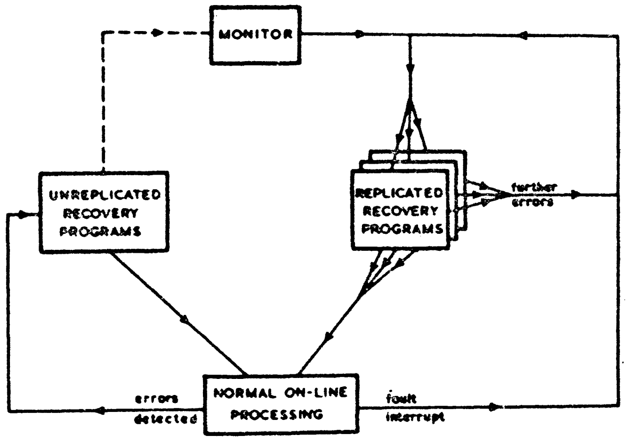
MONITOR
UNREPLICATED RECOVERY PROGRAMS
REPLICATED RECOVERY PROGRAMS
further errors
errors detected
NORMAL ON-LINE PROCESSING
fault interrupt

Fig. 8
Security System using a combination of replicated & unreplicated recovery programs

In the proposed implementation of System 250 the monitoring mechanism is made an implicit part of the unreplicated recovery programs. These recovery programs are activated by a process called the 'System Monitor'. This process runs at regular intervals and scans the system fault indicators. If any fault condition is detected then the appropriate recovery routine is activated. If persistent or multiple error conditions are detected then this implies that the fault is beyond the scope of the simple, unprotected, recovery programs, which are only intended to cope with relatively minor faults. In these cases System Monitor will force a fault interrupt, thus activating the second line of defence, the replicated recovery programs. This is illustrated in Fig. 9.

Of course it is important to protect the system against the possibility of the failure of this monitoring action. This can only happen in one of three ways:-

(1) the monitor can fail 'sane', detect that all is not well and force a fault interrupt.

(2) the monitor can fail 'dead', so that either it does not run at all, or does not perform any meaningful action when it does run.

(3) the monitor can fail 'crazy' so that it apparently runs correctly at regular intervals and yet does not respond to fault conditions.

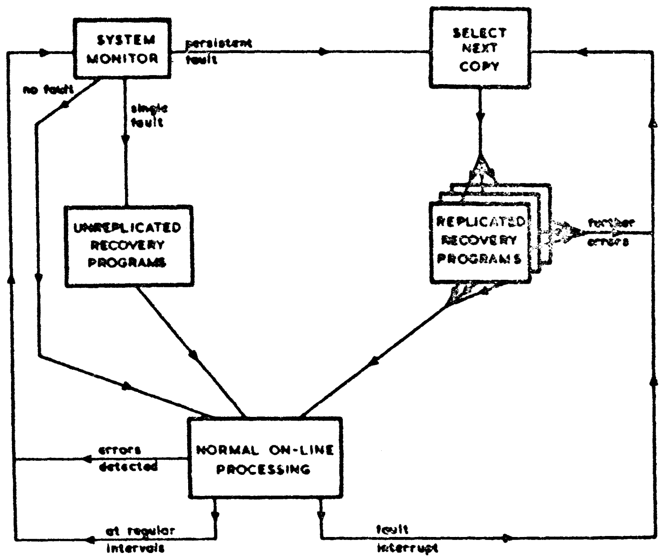

Fig. 9 .
Basic Structure of System 250 Security System

If the monitor fails 'sane' then the replicated recovery system is activated explicity by the monitor, and it can take effective action to recovery system operation. If the monitor fails 'dead' then an independent time-out mechanism is used to force a fault interrupt. This is equivalent to a periodic 'OK' signal which is used to reset a time-out, thus indicating that the system is operating correctly.

There remains the possibility that the monotor can fail 'crazy'. The probability of this happening can be reduced to any arbitrary level by incorporating sufficient self-checks into System Monitor, and ensuring that sufficient overlapping, independent software checks exist in the on-line system.

In general it is relatively easy to ensure that if the system fails then eventually, somewhere, one of the processors will generate a fault interrupt, thus activiating the replicated recovery programs.

There is one final modification which can usefully be made to the system illustrated in Fig. 9. It is fairly easy to ensure that even under the worst possible fault conditions at least one processor will generate a fault interrupt at some stage. Therefore the fault interrupt mechanism is used to ensure that the replicated recovery programs, and the associated powerful recovery actions, are activated when a major system collapse does occur. However, an isolated fault interrupt is symptomatic of nothing worse than a transient hardware fault, or simple software error. Ideally these should be dealt with by the unprotected programs, using recovery actions which cause minimum disruption to system operation.

Only repetitive or multiple fault interrupts should drive the system into the rather more drastic recovery measures adopted within the replicated recovery programs.

This feature can be incorporated fairly easily. After a processor has successfully completed the test program which forms the first part of the replicated programs, it places a message in a location which is scanned at regular intervals by System Monitor. When this message is detected, the other error indications are checked together with a fault count for the processor which generated the message. Provided that this is an isolated occurrence the monitor process passes capabilities to the faulted proessor which allows itto rejoin the on-line system. If this particular processor has suffered a succession of fault interrupts it is assumed that it either has an intermittent fault, or a solid fault which is not detected by the test program. In either case it is not passed the capabilities which allow it to rejoin the on-line system but is forced back into the test program.

If System Monitor does not respond to the message then the assumption is made that either System Monitor has failed or that multiple error conditions have occurred. In this case the processor accesses the replicated recovery programs (Fig. 10).

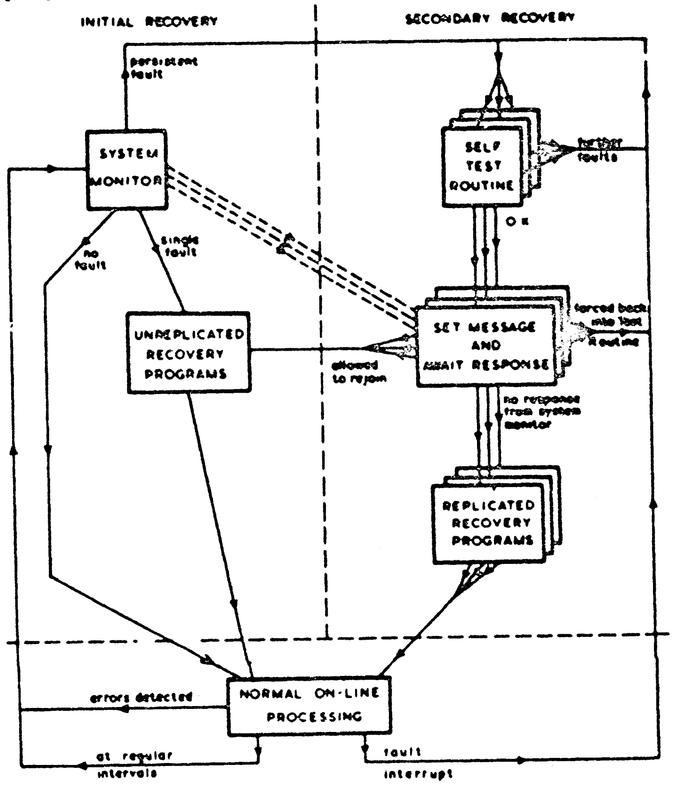

Fig. 10

Detailed Structure of System 250 Security System

Thus the general scheme is that in addition to the replicated programs which provide the basic level of recovery another group of programs is provided which run in the normal way as part of the basic supervisor. These programs form the first line of defence and provide a number of simple recovery actions which do not greatly disturb system operation. The general hierarchical structure shown in Fig. 3 is therefore extended by splitting the basic recovery system into two sections (Secondary Recovery and Initial Recovery). Only the essential kernal of this recovery system (Secondary Recovery) is replicated, and the rest (Initial Recovery) forms part of the basic supervisor (Fig. 11).

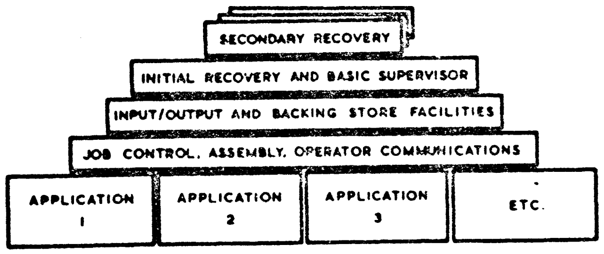
SECONDARY RECOVERY
INITIAL RECOVERY AND BASIC SUPERVISOR
INPUT/OUTPUT AND BACKING STORE FACILITIES
JOB CONTROL, ASSEMBLY, OPERATOR COMMUNICATIONS
APPLICATION 1
APPLICATION 2
APPLICATION 3
ETC.

Fig. 11
General Operation of the Basic Recovery System

So far we have discussed the sequence of actions to be taken under fault conditions, and the overall structure of the basic recovery system. Fig. 12 illustrates how these two are combined. Briefly, Initial Recovery which receives the initial error indications, is used to implement all the recovery actions which do not involve a complete system restart. The replicated programs of Secondary Recovery are used to provide the recovery actions which involve a complete system test and general restart.

To illustrate how this system reacts to the various kinds of failure which can occur it is useful to consider some specific examples.

First consider a software fault in one of the application areas. Typically, this type of fault causes programs to behave unpredictably when presented with certain, rather rare, combinations of data or timing circumstances. The error is detected either by the software checks within the application itself, or by hardware check circuits when the program involved attempts an illegal operation, such as writing into a read-only block.

If the error is detected by software then the response of the basic recovery system is to force a restart of the failed area. This action reconstructs data held in store and restarts processing new transactions, which is generally sufficient to clear the fault.

If the error is detected by hardware then, after various hardware test procedures, the particular transaction involved is restarted. This may be sufficient to clear the fault, but if it is not then subsequent faults will force an area restart.

In very rare circumstances the area restart may fail to clear the fault. This can only happen if the duplicate information held on backing store, which is used to reconstruct essential read write data, has been consistently corrupted in such a way as to cause further failures when it is used as part of the restart procedure. This type of fault is cleared by a subsequent recovery action in the sequence which involves a complete system reload, thus clearing any read/write data which has been generated by previous system operations.

Software faults in the operating system area, the basic supervisor for example, are dealt with in a similar way. However in this case the initial response to the error is more severe since an area restart involving any of the levels in the operating system will also imply a restart of all the application areas, rather than just the single application area involved as in the previous example.

Transient faults in processors or store modules have the effect or corrupting data, without permanently disabling a hardware unit, so that the immediate after-effects are indistinguishable from software faults. Thus the remarks made above also apply to this type of failure mode.

Consider the possibility of a processor fault. Recent trials on the system indicate that faulty processors usually take a fault interrupt very quickly after the incidence of a fault, within one or two milliseconds. Also recent tests have shown that the 'fault capture' level of the test program, which is obeyed by a processor after taking a fault interrupt, is very good, better than 99.5%. Thus the vast majority of processor faults will very quickly cause the faulty machine to take a fault interrupt. It is then trapped in the maze of the test program, which isolates it from the rest of the on-line system.

In general store faults will have an obvious and immediate effect on the system. Usually all the processors receive a parity fault indication very soon after the fault has occurred. This effectively disables the on-line system so that recovery is achieved through Secondary Recovery via a general test of the system and a complete warm start.

Hardware faults which are not located by the test routines, either because they are intermittent or beause the test routines are not comprehensive enough, are difficult to recover. They may be located by means of fault counts, or in the case of intermittant faults by repetitive use of the hardware test programs. However if none of these mechanisms do locate the fault then the final, last ditch, action taken by the recovery system is to attempt to find a viable configuration by means of trial recongiguration. How quickly this is achieved depends on the nature of the fault. If the fault is seriously affecting system operation, so that its effects can be detected very earily, then a medium sized system can work through all possible combinations of the central control equipment in something like two minutes. If the fault only causes the failure of the occasional transaction then the system is performing useful work. Provided the reduced performance is acceptable then the automatic recovery mechanisms will not be activated at all, since the system is, to all intents and purposes, operating satisfactorily. This type of non-urgent fault will eventually be cleared by the maintenance engineers, who receive information regarding all error indications recorded.

References

1. K. Hamer-Hodges 'Fault Resistance and Recovery within System 250' - Presented at this conference.

2. D. C. Cosserat 'A Capability Oriented Multi-Processor System for Real-Time Applications' - Presented at this Conference.

3. E. Djikstra 'The Structure of 'THE' Multi-programming System" Com. A.C.M., Vol. 11, No. 5 May, 1968, pp. 341 - 346

Acknowledgement

I would like to thank the many colleagues on whose work this paper is based and the Directors of the Plessey Company for permission to publish it.

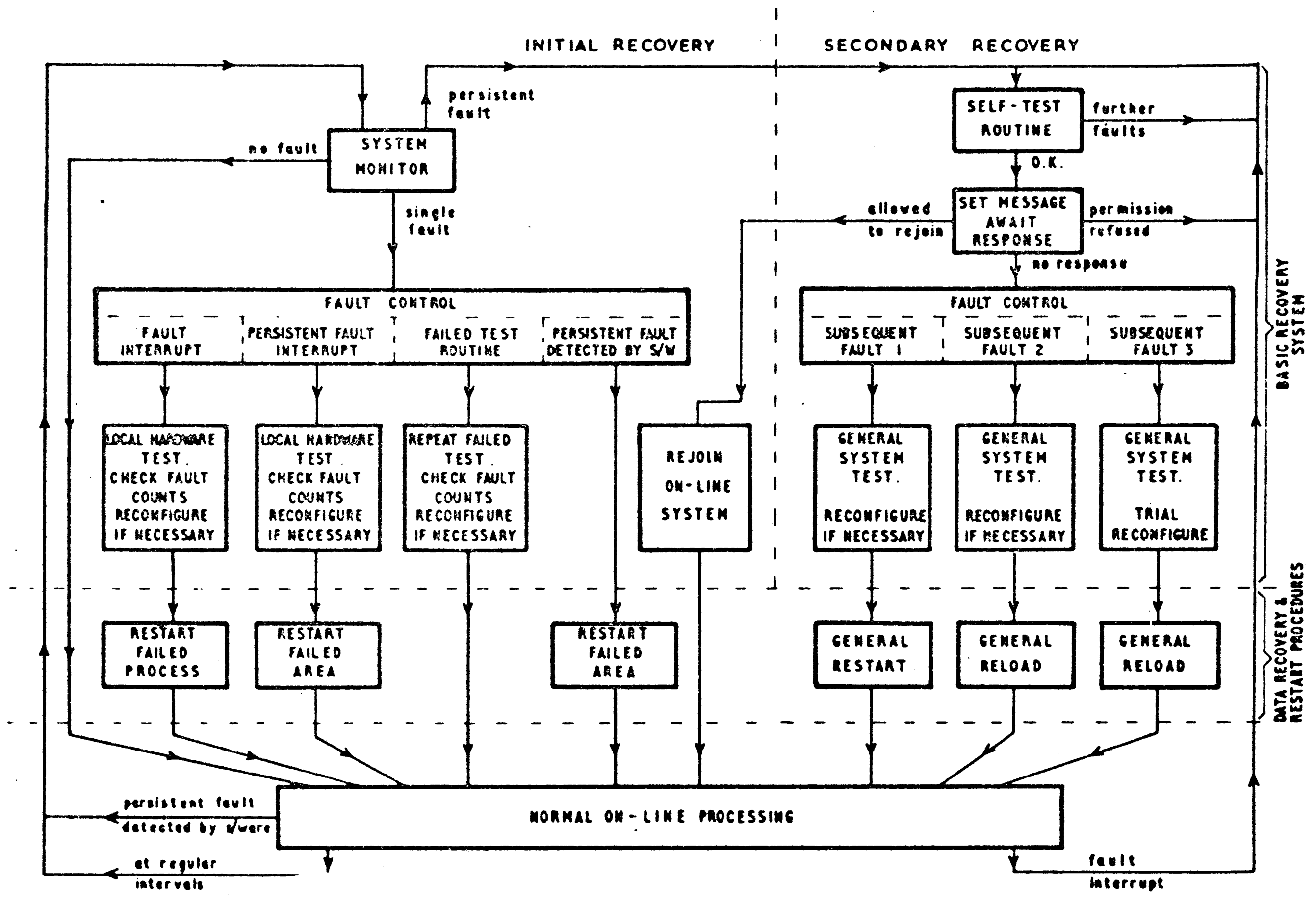
INITIAL RECOVERY
SECONDARY RECOVERY
persistent fault
no fault
SYSTEM MONITOR
single fault
FAULT CONTROL
FAULT INTERRUPT
PERSISTENT FAULT INTERRUPT
FAILED TEST ROUTINE
PERSISTENT FAULT DETECTED BY S/W
LOCAL HARDWARE TEST. CHECK FAULT COUNTS RECONFIGURE IF NECESSARY
LOCAL HARDWARE TEST. CHECK FAULT COUNTS RECONFIGURE IF NECESSARY
REPEAT FAILED TEST. CHECK FAULT COUNTS RECONFIGURE IF NECESSARY
RESTART FAILED PROCESS
RESTART FAILED AREA
RESTART FAILED AREA
SELF-TEST ROUTINE
further faults
O.K.
SET MESSAGE AWAIT RESPONSE
allowed to rejoin
permission refused
no response
REJOIN ON-LINE SYSTEM
FAULT CONTROL
SUBSEQUENT FAULT 1
SUBSEQUENT FAULT 2
SUBSEQUENT FAULT 3
GENERAL SYSTEM TEST. RECONFIGURE IF NECESSARY
GENERAL SYSTEM TEST. RECONFIGURE IF NECESSARY
GENERAL SYSTEM TEST. TRIAL RECONFIGURE
GENERAL RESTART
GENERAL RELOAD
GENERAL RELOAD
BASIC RECOVERY SYSTEM
DATA RECOVERY & RESTART PROCEDURES
NORMAL ON-LINE PROCESSING
persistent fault detected by s/ware
at regular intervals
fault interrupt

Fig. 12

Implementation of System 250 Security System

STRUCTURE AND INTERNAL COMMUNICATIONS OF A TELEPHONE CONTROL SYSTEM

J. Crompton

Plessey Co. Ltd.
Liverpool, England.

Summary

Current plans for the introduction of computer controlled telephone exchanges to Great Britain envisage the formation of a new telephone network which will interwork with the current network and ultimately replace it. The basic module of the new network is a Switching Unit, which is controlled by a Processing Utility. Switching Units are composed of a number of subsystems, and each subsystem is subject to standard definitions both for its interfaces and for the function it performs. The hardware/software ratio of each subsystem is at the discretion of the manufacturer, but subsystems can be regarded conceptually as having a hardware component and a software control component. The action of the subsystem control programs is coordinated by a further control program, and a great amount of interaction is necessary between these programs during the setting up of a telephone call. The software mechanisms necessary for internal message handling and process creation must be chosen with great care bearing in mind the various trade-offs possible, processor utilization, and the definitions of the subsystem standard interfaces.

Introduction

Development of the British Telephone Network is guided largely by the Advisory Group on System Definition (AGSD) - a body consisting of representatives from both the Administration (British Post Office) and from the various manufacturers of telephone equipment. Any future computer controlled telephone exchange which is to be used in the United Kingdom will be subject to constraints laid down by AGSD.

The concept currently proposed by AGSD is to form an "overlay" network of Stored Program Control (SPC) exchanges. By this is meant a system which could start off life in a very small way - possibly a single exchange - interworking with the existing telephone network, and which could then grow in discrete stages. This would form a new, small network of SPC exchanges, which interfaced with the old network at selected points. As the new overlay network grows, it will slowly replace parts of the old, until eventually the entire system will consist only of SPC exchanges.

System Structure

Switching Units

The basic module of this new network is known as a Switching Unit. Switching units are of several different types, and each type can have many different designs and constituent elements. Basically, the function of each switching unit is to provide facilities whereby various telephone circuits can be monitored and interconnected under the instructions of a centralized control. This control may be located with the switching unit, but equally may be remote and operate via a data link. The centralized control is known as a Processing Utility. The two most common types of switching unit are:-

1. Subscriber Switching Unit
The Subscriber Switching Unit interfaces directly with the telephone user, by means of wires from the subscriber's premises. Fig. 1 indicates schematically a subscriber switching unit, which, with its interface to the existing network, could provide the start of the new network.

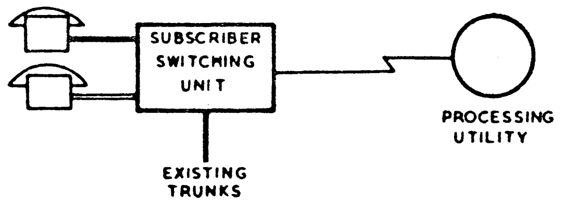

Fig. 1

TYPICAL INTERCONNECTION OF SWITCHING UNITS

2. Main Switching Unit
The Main Switching Unit is normally connected to a number of subscriber switching units, and also may be connected to other main switching units. This is indicated schematically in Fig. 2.

Although figures 1 and 2 show the switching units as connected by a single data link to a single processing utility, more complex arrangements will apply in practice, for security reasons.

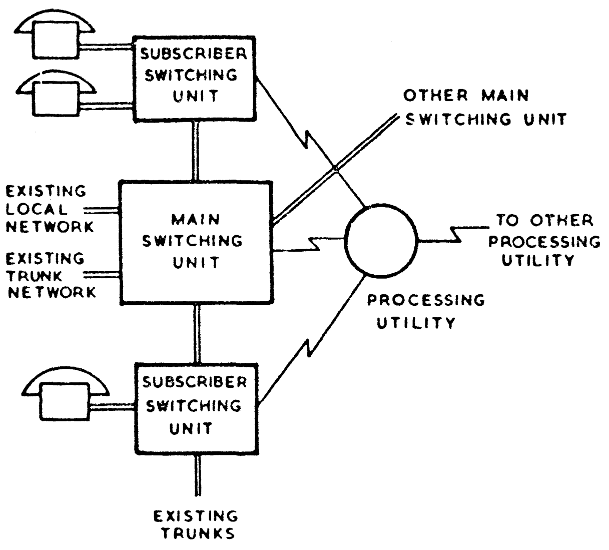
SUBSCRIBER SWITCHING UNIT
OTHER MAIN SWITCHING UNIT
EXISTING LOCAL NETWORK
MAIN SWITCHING UNIT
EXISTING TRUNK NETWORK
TO OTHER PROCESSING UTILITY
PROCESSING UTILITY
SUBSCRIBER SWITCHING UNIT
EXISTING TRUNKS

Fig. 2

MAIN SWITCHING UNIT

Subsystems

Each of these switching units is composed of a number of distinct elements,known as subsystems. Subsystem units are so chosen to provide interfaces which can be rigidly defined, and remain constant between equipment manufacturers, enabling equipments of various designs to interwork satisfactorily. Each subsystem performs a distinct function within its interface boundaries; it is the declared intention of AGSD to define these functions and interfaces. Some typical functions which can readily form subsystems, however, are:-

1. Subscribers Subsystem
This subsystem provides the complete interface between a particular group of subscribers and the rest of the network. It provides all system communication with the subscriber - for example it will provide dial tone and busy tone to the subscriber, and will accept dialled or keyed digits from him. The subsystem also performs some switching and concentration of subscribers lines.

2. Transit Subsystem
This subsystem provides a switching facility, and thus permits different subsystems to be interconnected and cross connected as desired.

3. Interface Subsystem
This subsystem is used to connect the new network with the old. It must provide all types of signalling and all facilities in use on the particular existing junctions with which it is connected.

4. Manual Board Subsystem
This subsystem provides the second "human" interface into the system (the first being the subscriber). The subsystem must provide all facilities necessary for operational staff to provide assistance to subscribers, monitor and test lines etc.

5. Miscellaneous Terminations Subsystem
This subsystem contains the various devices required by the administration - such as time announcement machines, message recorders, facilities for interconnecting multi-subscriber calls.

The configuration of Fig. 2 is redrawn in Fig. 3 to show some subsystems which could typically be involved, and the ways in which they could interface with each other.

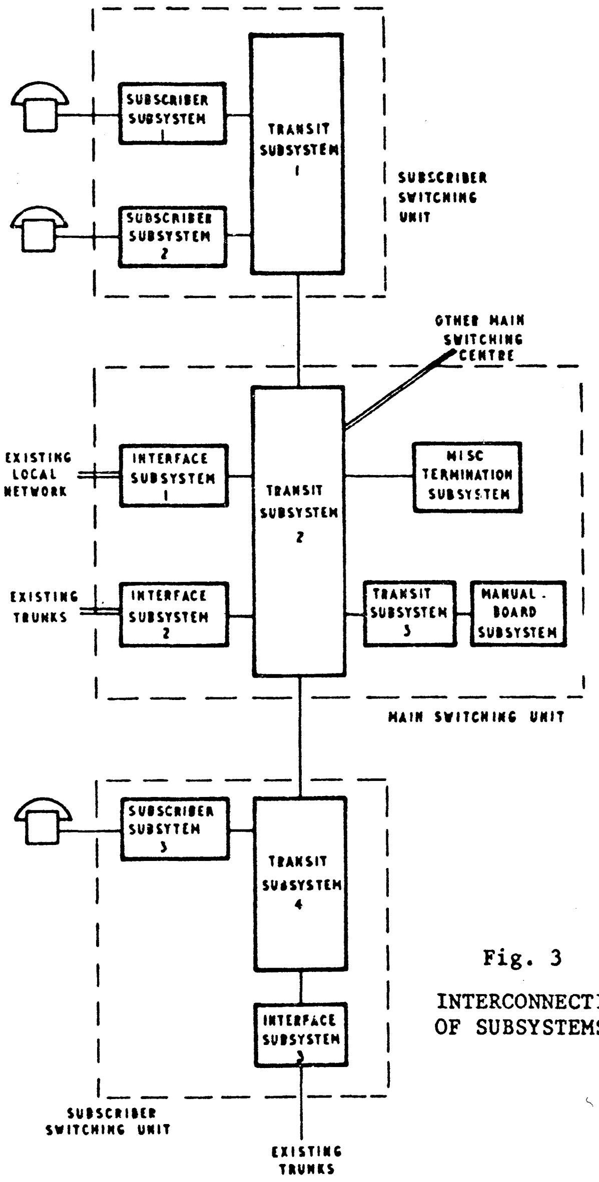

Fig. 3

INTERCONNECTION OF SUBSYSTEMS

Subsystem Structure

Subsystems are chosen to perform particular functions within defined interfaces; the manner in which the functions are performed will depend upon the method of implementation chosen by the particular manufacturer. This detailed implementation can vary greatly - not only between manufacturers, but within manufacturers as technologies advance. In particular, the amount of work performed by the hardware and the amount performed by the software can vary. For example, consider the hardware/software trade-offs which are possible in the design of the switchblock part of a subsystem:-

The basic requirement is to connect one particular input from a group of inputs, to a particular output. Fig. 4 shows a group of 12 inputs and 6 outputs, and a possible method of performing the connections by two stages of switching - each point marked X represents a switch or "crosspoint". It can be seen that by judicious operation of two crosspoints, any input can be connected by one of several paths to any output - provided that the paths are not already in use for another connection. Several methods of arranging this connection are possible: for example:-

1. Use of "intelligent" hardware, which would accept the identities of the two terminals to be connected, effect the connection if possible, and then return a "success" or "fail" message. This solution requires a minimum of software.

2. Use of simple hardware, which would merely activate or deactivate any nominated crosspoint, as instructed. This solution requires all the work to be done by the software - even to the extent of keeping a "map" of the crosspoints, in which busy ones are marked, and from which an available path can be selected, details of which are sent to the hardware.

3. Use of hardware falling between these extremes for example hardware which would activate and de-activate nominated crosspoints, check and report upon the success of the operation, and also provide facilities for the software to interrogate the state of nominated crosspoints. This solution leaves the "intelligence" with the software, but provides security for the current details of crosspoint settings.

4. Use of solution 2. or 3. above, but placing the necessary software in a local mini or micro processor, which acts upon instructions received from the processing utility.

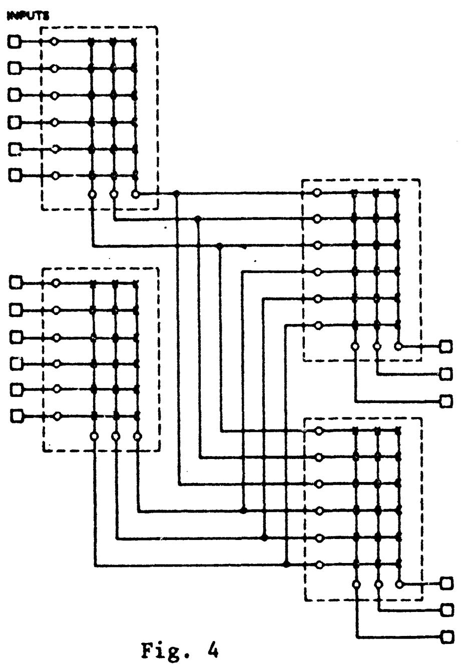

Fig. 4

TYPICAL 2-STAGE SWITCHING

Since it is possible for any subsystem to contain software, it is logical to consider each subsystem as consisting of two interdependent parts - the hardware, and the software within the processing utility. Any program structure, therefore, will conceptually contain a number of distinct subsystem control programs, but in the limit of complex hardware or local mini-processor implementation, the control programs will be simple message handlers.

These subsystem control programs must be able to transmit and receive messages to and from their hardware counterparts. The physical means of this message transmission may include a data link, and most probably will include methods of multiplexing and de-multiplexing along some message transmission medium - for example the normal I/O handling software of the virtual machine in the processing utility could interleave messages for different subsystems along a single highway. The content of these messages is private between the hardware and software parts of the subsystem (though the format may be affected by the communications medium); the transmission means should ideally be transparent. The interposition of additional hardware and software between the subsystem hardware and its control program in order to resolve message addressing and transmission problems in no way violates the concept of the subsystem with defined interfaces; it merely provides a transparent interconnection. Figure 5 shows schematically the type of arrangement that could exist for the system depicted in Figure 3. Each hardware subsystem has its software counterpart in the processing utility. The points marked X indicate interfaces which are likely to be defined as AGSD standard interfaces - these are interfaces at the software end of the subsystems; other interfaces subject to definition lie at the hardware end at the human interfaces.

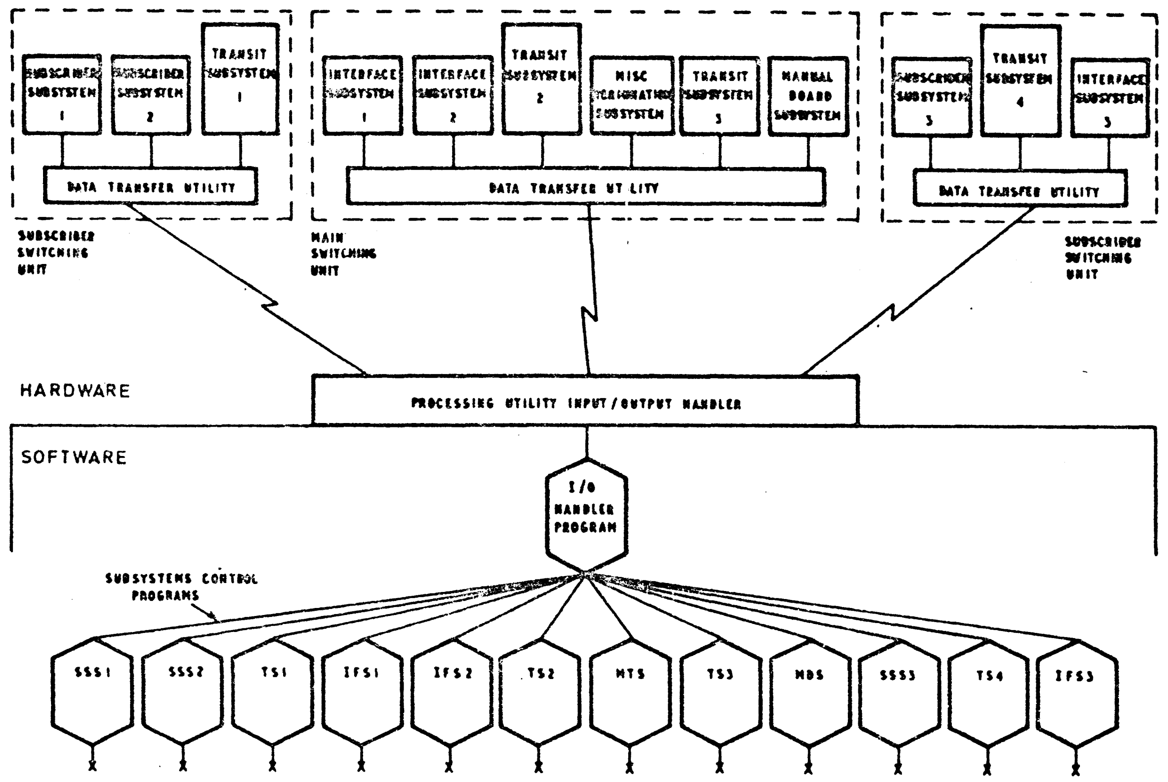

Fig. 5 TYPICAL INTERCONNECTION OF HARDWARE AND SOFTWARE SUBSYSTEMS

In practice the arrangement will probably differ from that shown in Figure 5. Frequently a single manufacturer will be responsible for a complete switching unit - if not for several co-located switching units. Two or more similar hardware subsystems could be controlled by a single control program, operating upon several data bases. Also, economics could dictate that certain pieces of equipment be shared by several subsystems. The hardware part of each subsystem consists of several devices, which are each treated as peripheral units. One of these equipments for example, which is often expensive, is called a Marker. The marker is the peripheral which controls the operation of crosspoints in the switchblock, as explained previously. A marker may well have the capacity to control more than one switchblock, so its costs could be shared among subsystems located together. Figure 6 indicates a possible method of re-structuring the software configuration of Figure 5. The handling programs for the individual peripherals (such as the marker) are shown, and these programs communicate with the subsystem control program proper, which must co-ordinate the operation of the peripherals in its subsystem hardware. Only one subscriber subsystem control program is shown - this will handle all three subsystems from three separate data bases; similarly for the other subsystems.

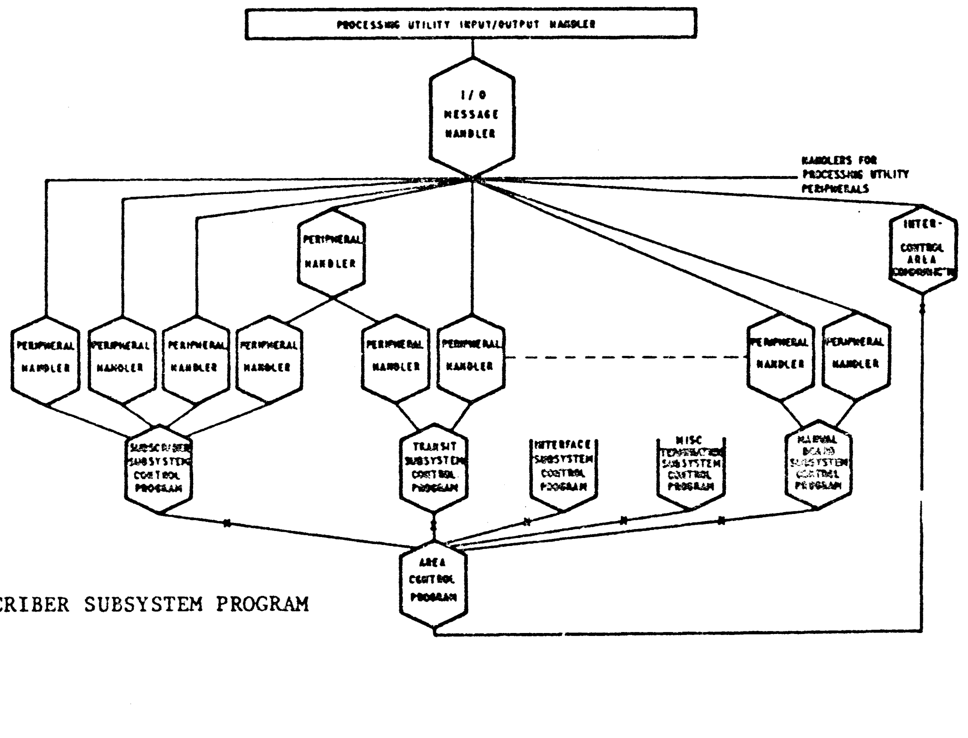
PROCESSING UTILITY INPUT/OUTPUT HANDLER
I/O MESSAGE HANDLER
HANDLERS FOR PROCESSING UTILITY PERIPHERALS
INTER-CONTROL AREA COMMUNIC'N
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
PERIPHERAL HANDLER
SUBSCRIBER SUBSYSTEM CONTROL PROGRAM
TRANSIT SUBSYSTEM CONTROL PROGRAM
INTERFACE SUBSYSTEM CONTROL PROGRAM
MISC TERMINATING SUBSYSTEM CONTROL PROGRAM
MANUAL BOARD SUBSYSTEM CONTROL PROGRAM
AREA CONTROL PROGRAM

Fig. 6

TYPICAL SUBSCRIBER SUBSYSTEM PROGRAM

A software structure is now starting to emerge. A number of software "modules" have been identified, and some have software interfaces which are the subject of future definitions The word module is used in the sense of a self-contained piece of software, which could be written and tested in isolation. As yet, no means of co-ordinating the operation of these modules has been mentioned; it is here that the concept of a Control Area is introduced. A Control Area consists of a group of switching units which are controlled by the same processing utility, and within which it is possible to select overall the path that will be used by a particular call, before that path is set up. In Figure 6, a software module called Area Control has been introduced, and this co-ordinates the operation of the individual subsystem control programs. The area control program can be regarded as holding the intelligence for the call, and the subsystems execute specific commands given by area control.

Internal Communication

Consider the type of interaction that will be necessary between the software modules of Figure 6 - for example when a subscriber in subscriber subsystem 1 wishes to make a call to a subscriber in subscriber subsystem 3 (Figure 3). The first indication that a call is to be made is given by the subscriber lifting off his handset; this event is detected by a peripheral called a Subscribers Line Circuit within the subscribers subsystem. The event will pass via the I/O message handler, peripheral handler and subsystem control program to the area control program which must examine its records to determine the type of service permitted to the particular subscriber. If dialling out is permitted, the subscriber subsystem will be instructed to connect the appropriate type of digit receiver, in anticipation of keyed or dialled digits, to connect any supervisory circuits that may be required, and to connect dial tone to the line. As digits are received, they pass via the chain to the area control program until eventually sufficient digits have been received to determine the destination of the call. After checking the availability and status of the called subscriber, instructions must be provided to the appropriate subsystems at the apprópriate times to provide ringing current to the called subscriber, ring tone to the calling subscriber, to remove these conditions, to set up a path via the two transit subsystems, to check upon the continuity of the path - and eventually to clear down all connections. Should the destination lie in a different control area, messages must be sent either directly to the destination control area or to some intermediate (transit) control area which must itself activate appropriate subsystems.

It can be seen then, that within the software structure which has now emerged, there is a great requirement for the passing of messages between individual modules. Careful study must be given during system design to the mechanisms that will be involved in message handling, and the closely allied topic of process creation. It is assumed that all modules are written in a re-entrant manner, so that conceptually one or many processes using a particular module may be in existence simultaneously. The term "process" is here used in the dynamic sense, to mean the serial execution of the code in a module or program. A process may be associated with a particular module, or it may be associated with a message, for example, and cross module boundaries.

Figure 6 shows that each module contains a discrete number of "message ports" or, in other words, has a number of interfaces across each of which particular types of message could be expected. An extremely simple mechanism could be to place at each such input port a message queue. This queue is loaded by the output port, which generates the message, calling upon a common, centralized queue loading mechanism. The call to the queue loader specifies the name of the wanted queue, and the loader locates the queue by using a close association with the space allocation mechanism. (Absolute addresses could not be used, because in a system of this nature with a requirement of many years mean time between failures, it is necessary to move programs and data around, when system components either fail or are released for scheduled maintenance). Once one of these modules is scheduled, it runs until all its queues have been emptied, at which point it terminates. The time scheduling algorithm can be constructed to any arbitrary degree of complexity. This system has several attractions - the time and space overheads involved are quite small, and there is no danger of messages getting out of sequence and "jumping their queues". Also no contentions arise for file access; since only one process exists on any particular module at any instant access to that module's in-core data bases need not be subject to any lock and key control.

This system, however, could become quite inefficient as traffic grew. The number of processes is equal to the number of software modules - but it is quite probable that the processing utility consists of a number of processors working in parallel in a load-sharing manner. The precise number of processors is of no interest to the applications programs, provided that collectively the processors provide sufficient processing power; the supervision and co-ordination of the individual processors can be regarded as a function of the "virtual machine". The net result, however, is that several processes may be able to run concurrently. The system of one process per module forces all telephone calls to queue for sequential service by the area control program, whereas logically there is no reason why separate calls should not be processed in parallel by use of several processes on several processors - thus removing what could become a serious bottleneck on processor time utilization as traffic increased.

Such considerations lead to the proposal of creating a separate process for each telephone call. As this process completes execution of one module, it transfers control to the next module required by the particular call. Data associated with a particular call is carried in the "process base", or workspace associated with the process, and this reduces the overhead of message passing between processes. A number of difficulties are found with this approach, however. All messages entering the system require a certain amount of processing before thay can be associated with a particular call, and it is only when a message has travelled a certain distance that it can be picked up by its parent process - and this distance will depend upon the point at which the parent process last suspended. Parallel processing of certain activities associated with a call is not possible for example, in the subscriber call described previously, the parallel actions by the two subscriber subsystems of setting up ring tone and ring current would need to be carried out sequentially (although of course each subsystem could be simultaneously active upon different calls). Even more serious difficulties are encountered when a call requires to be handled by a second (or third) control area - it is not feasible for the process to cross control area boundaries. This approach also entails a process crossing a subsystem "standard interface" which ideally should be defined in terms of messages only.

Yet another possibility is to use a process per module per telephone call. This approach requires a message passing mechanism which can deal with a high message rate, and which can associate messages wuth processes. The combination suggests a centralized system which uses semaphores for communication, and which is intimately associated with the time scheduler. The system must also allow information to be passed with each semaphore. Such a general mechanism is currently being implemented by one telephone manufacturer; it is conceptually simple, and flexible in application, permitting easy system expansion.

The mechanism readily provides association of messages with processes, and provides for reactivation of suspended processes. Great care must be exercised in its use, however, because of the space and time overheads inherent in such an approach. Even though it provides a useful mechanism for the process per module per telephone call. Some problems still remain with this approach - such as the file locking problem mentioned previously. For example, in a subsystem which contains a switchblock, and which maintains a store image of that switchblock, each call which uses the image to select a path will require unique access to the image for duration of the path choice algorithm; some method of constraint is necessary if a process per module per telephone call approach is used.

Obviously, none of these systems represents black or white; all are shades of grey. An attractive compromise is to treat different types of module in different ways. The I/O message handler is really a function of the processing utility, but can be considered here as being in two parts; input, and output. The input part is a single process, which is scheduled at regular intervals, and once scheduled runs cyclically until all incoming messages have been handled. As each message arrives, the addresses and other information from the Data Transfer Utility will identify the device from which the message originated, and the message is then passed to the appropriate peripheral handler (by the semaphore mechanism), where a process is activated for the particular message. The output part is also a single process, which is activated whenever a message is sent to it via a semaphore, from any peripheral handler program; having dealt with the message, the process suspends itself awaiting the next message.

The peripheral handler processes, in the case of input, will "funnel" down to a single process, which runs the subsystem control program. This funnelling can be achieved by private queues, in the case of unshared peripherals, or by use of the semaphore mechanism for peripheral handlers shared between subsystems. The single subsystem process can now service its messages in cyclic fashion, and has no contention problems for its files. In the case of subsystem control programs controlling several different subsystems, each on its own database, there is one process per database.

At control area level, yet another arrangement applies. Communication with the subsystem control programs is handled by the semaphore mechanism, thus maintaining a message interface. Each instruction given to the subsystem is accompanied by a "tag" which uniquely identifies the particular call (for example a call number) and this tag is later returned by the subsystem when reporting upon the action performed. Within the area control program, a process is initiated which handles the originating part of all calls. When the initial message comes from the subscribers subsystem indicating that the handset has been lifted, this Originating Call Process allocates the unique tag to the call and handles the early parts of call set up. This single process limits itself to handling a fixed number of calls the number is dependent upon the structure of the processing utility; once the number is exceeded, a further parallel process is created to handle subsequent calls (thus ensuring equitable use of processing resources). Once sufficient is known about the destination of the call, a second process is created. This process may be in the same control area or a different control area, depending upon the destination of the call - creation of the second process, however, ensures that all calls can be treated in standard manner, whether they be inter or intra control area. The second process may be a Terminating Call Process, but the originating and terminating call processes can be separated by one or more Transit Call Processes, if the call needs to pass through several control areas. Each such process runs in a cyclic manner, and will itself create further processes as the traffic load increases. All these processes existing in the area control program communicate with the subsystems by the semaphore mechanism.

Conclusions

This approach is by no means the only possible, but it does illustrate the type of mechanisms which are necessary for the organizing of intercommunication between various parts residing inside an SPC computer system. Most of the work currently being performed in this area is subject to change, since the definition of the interfaces is not yet available. As detailed implementation of SPC progresses, however, manufacturers are solving problems in increasing detail, thus permitting interface and functional definitions to be arrived at which are both efficient and enduring.

Acknowledgement

The author wishes to thank his many colleagues upon whose work this paper is based and the Management and Directors of The Plessey Company for permission to make this information available.